تعلم بدون تعقيد ..

JSP

JAVA Server Pages



المركز الرئيسي : 11شارع د/محمد رأفت – محطة الرمل – الإسكندرية
تليفون وفاكس : 4838326 (03)(+2)
موبايل : 0101634294 (+2) - 0123357844 (+2)
Email: info@egyptbooks.net
URL: www.egyptbooks.net

اسحاق بخيت
(embekheet@yahoo.com)



وقل اعملوا فسيرى الله عملكم ورسوله والمؤمنون



طبعة سبتمبر 2005

رقم الإيداع

2005/14390

ISBN

977-17-2417-7

مقدمة

تستخدم لغة JSP بهدف إنشاء صفحات ويب تفاعلية معقدة بطريقة سهلة فتعطى هذه اللغة الكثير من الإمكانيات التي تمكنك من إدراج بيانات في الصفحة بطريقة سهلة ويمكنها الاتصال بجميع أنواع قواعد البيانات وأيضا استيراد بيانات من ملفات XML ولا يتوقف الأمر على ذلك فقط ولكن أيضا يمكنك إنشاء أدوات برمجية Components تقوم بعمل معين غير متاح في اللغة بحيث تستطيع استخدامه أكثر من مرة.

لمن هذا الكتاب؟

هذا الكتاب مناسب للمبرمج المبتدئ الذي يريد أن يعرف لغة JSP ولكن يجب أن يعرف القارئ لغة HTML والتعامل مع النماذج Forms وأيضا سبق له التعامل مع أي لغة تتبع أسلوب البرمجة المتجهة Object Oriented Programming ويفترض أن يعرف برمجة التعامل مع الملفات .

الفصل الأول

البدايــة

ماذا تحتاج لكي تبدأ ؟

- ✓ جهاز كمبيوتر به نظام التشغيل Windows XP أو أعلى (إذا كنت تستخدم windows 2000 فتأكد من تحميل service pack الخاصة به).
- ✓ تحميل برنامج Apache Tomcat من الوصلة الآتية :
http://jakarta.apache.org/tomcat ، حيث يقوم هذا البرنامج بمثابة المترجم لأوامر JSP ويوجد غيره من برامج الترجمة مثل WebSphere من شركة IBM إلا أن برنامج Tomcat متاح مجانا ومفتوح المصدر.
- ✓ مكتبة الجافا JDK وهى المكتبة التي يقوم باستخدامها برنامج Tomcat لتنفيذ أوامر JSP ويمكن تنزيلها من http://java.sun.com ويوجد الإصدار 1.5 فإذا كان هناك إصدار أحدث يفضل تنزيله كما يجب تنزيل النسخة الكاملة من JDK وليس JRE (Java Runtime Environment) لعدم وجود أدوات كافية بها لأمثله الكتاب وآخر حجم للمكتبة الكاملة كان تقريبا 105 MB .

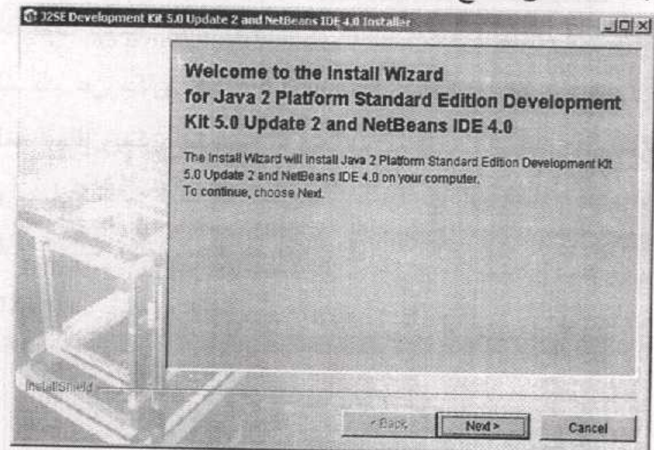
تحذير: لا يمكنك استخدام الإصدارة ***5.5*** من برنامج المزود ***TOMCAT*** إلا مع أصدارة ***JAVA 1.5*** أو أعلى فلن يتم تحميل مكتبة ***TOMCAT*** مع استخدام الإصدارات السابقة مثل ***1.4*** .

تركيب برنامج J2SE 5 Update 1:

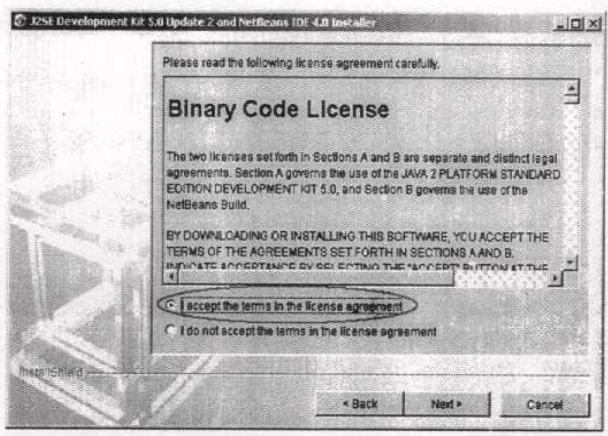
- قم بالضغط المزدوج على أيقونة برنامج التركيب .


jdk-1_5_0_02-nb-4_0-ml-win.exe

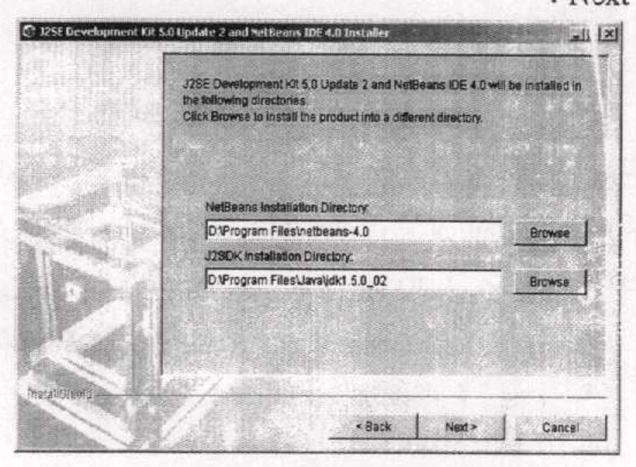
- سيبدأ البرنامج بالعمل ، أنتظر قليلا حتى تظهر شاشة التركيب الأولى التي يكتب بها ترحيب بسيط لإنزال JDK 5 Update 2 قم بالضغط على مفتاح Next.



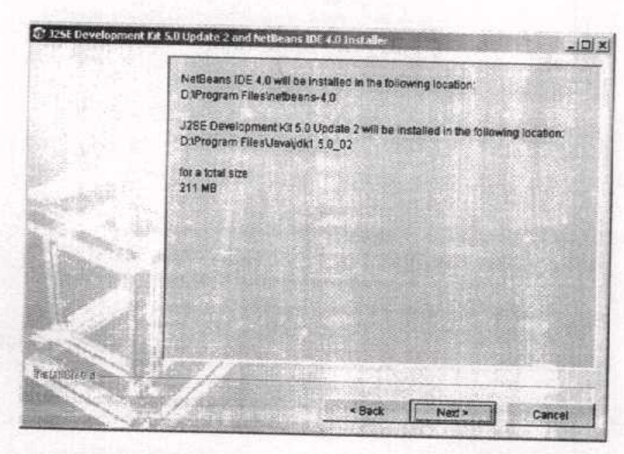
- تظهر بعد ذلك نافذة الترخيص التي تسألك إذا كنت توافق على اتفاقية الترخيص ويجب أن تختار I accept وإلا لن يتم استكمال تركيب البرنامج ، قم الآن بالضغط على I accept ثم المفتاح Next للانتقال إلى الشاشة الثانية.



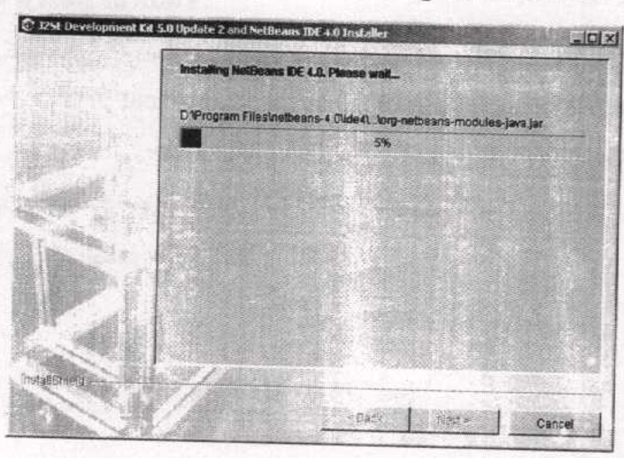
- إذا قمت بتحميل ملف JDK + NetBeans IDE 4.0 تظهر نافذة لتسألك عن مكان تحميل هذا البرنامج الإضافي وهو من محررات لغة الجافا ويمكن استخدامه لكتابة كود الجافا به وترجمه هذا الكود ، وأيضا اكتشاف الأخطاء كما يسألك البرنامج عن المكان الذي تريد فيه تركيب مكتبة الجافا قم بقبول المسار الافتراضي وأضغط على Next .



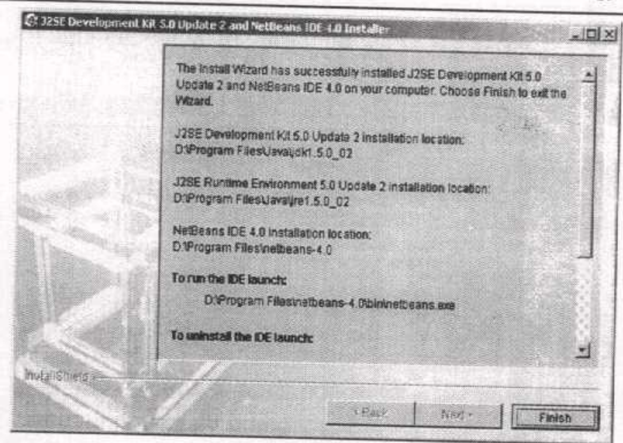
- تظهر رسالة بعد ذلك لتأكيد عملية التنزيل أضغط Next



- وانتظر شريط التحميل حتى ينتهي وبذلك يكون جهازك قادر على تشغيل وترجمة برامج الجافا.



- ثم أضغط أخير على مفتاح الانتهاء Finish ويجب إعادة تشغيل الكمبيوتر وهي خطوة ضرورية حتى يكتمل التثبيت .



بهذا تكون قد أكملت خطوات تركيب مكتبة الجافا والآن يجب عليك تركيب مزود صفحات JSP وهو برنامج Tomcat.

تركيب برنامج Tomcat:

- يمكنك تنزيل برنامج Tomcat من الوصلة :

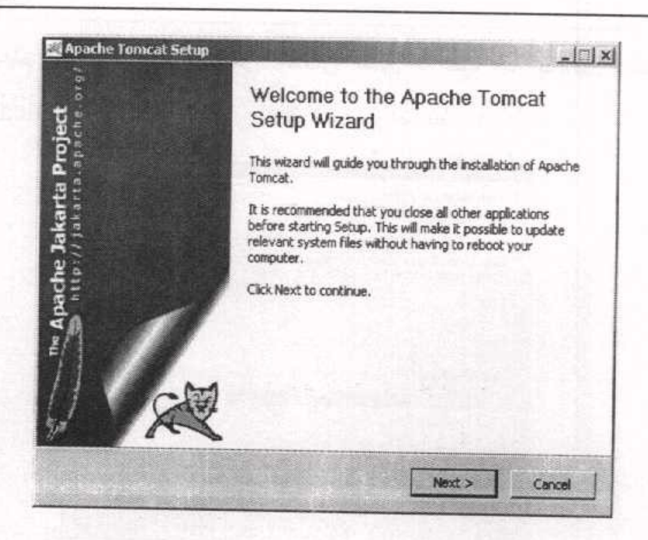
http://apache.secsup.org/dist/jakarta/tomcat-5/v5.5.8-alpha/bin/jakarta-tomcat-5.5.8.exe.مباشرة

بعد تنزيل البرنامج سوف تشاهد أيقونة كما بالشكل للإصدار Tmocat 5.5 فإذا وجدت إصدار أحدث يفضل تنزيله.

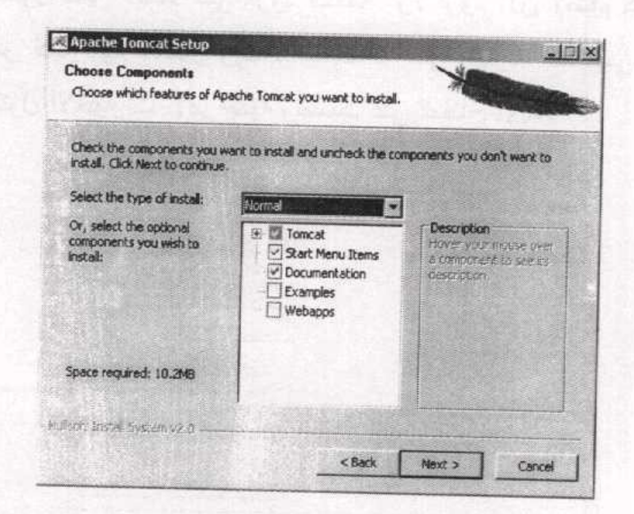


jakarta-tomcat
-5.5.8.exe

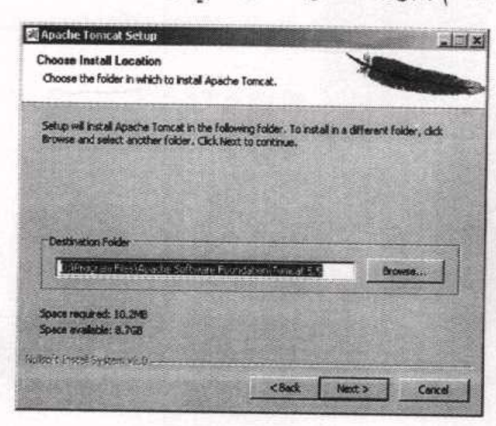
- قم بالضغط المزدوج على الأيقونة وسيتم بدء التركيب مع نافذة للترحيب قم بالضغط على المفتاح Next للانتقال إلى النافذة التالية.



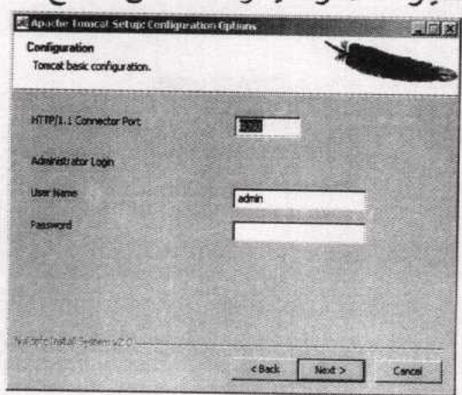
• توجد بالنافذة التالية اختيارات التركيب ومن خلالها يمكنك اختيار تنزيل ملفات إضافية أم لا مثل ملفات الأمثلة وملفات المساعدة كما بالشكل التالي ، قم بقبول الاختيارات واضغط على المفتاح Next .



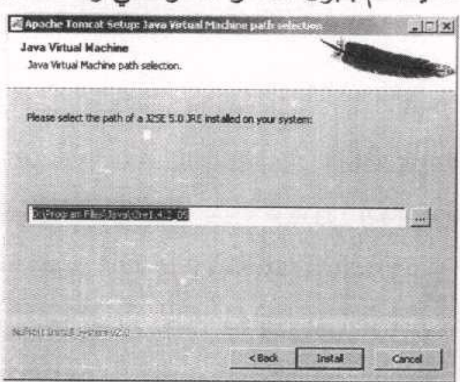
- سيتم بعد ذلك سؤالك عن المكان الذي ترغب فيه بتنزيل نسخة Tomcat قم بقبول المسار الافتراضي وأضغط Next.



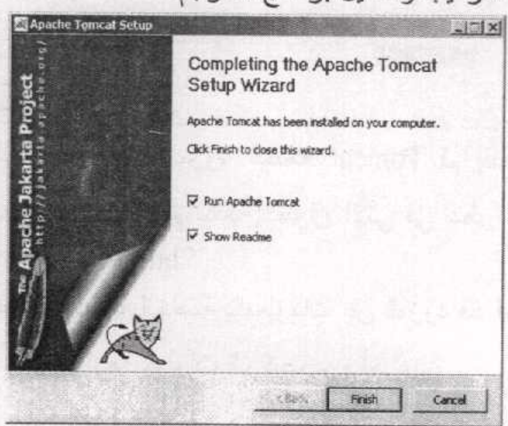
- بعد ذلك سيتم سؤالك عن رقم الميناء الخاص بتنفيذ تعليمات JSP وهو المنفذ 8080 الخاص بتنفيذ تعليمات البرتوكول http ويفضل عدم تغييره ، ثم يسألك عن اسم المستخدم وكلمة السر الخاصة بدخول مدير النظام على مزود الخدمة ولا لزوم الآن لإنشاء كلمة سر لأننا نقوم باختبار وإنشاء الصفحات على جهازك الشخصي قم بقبول الاختيارات الافتراضية واضغط على المفتاح Next .



- سيقوم بعد ذلك برنامج التركيب بالبحث عن مكتبة الجافا وإذا وجدها سيقوم بكتابة المسار الخاص بها في النافذة التالية ويسألك إذا كان هذا المسار صحيحا قم بقبول المسار الافتراضي وأضغط Next .

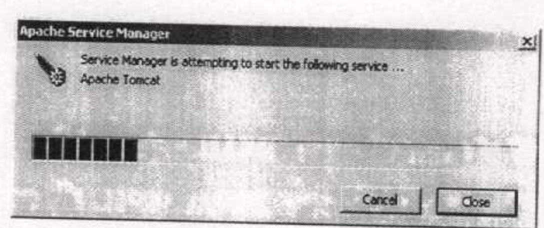


- بعد ذلك يتم تنزيل ملفات برنامج المترجم Tomcat وتظهر نافذة الانتهاء التي تعطيك اختيارين يتم تنفيذهم عند الانتهاء من التركيب وهم تشغيل برنامج Tomcat وفتح ملف للتعليمات المساعدة قم الآن بإزالة الاختيار show readme ثم أضغط على المفتاح Finish لإنهاء التركيب وتشغيل برنامج المترجم.

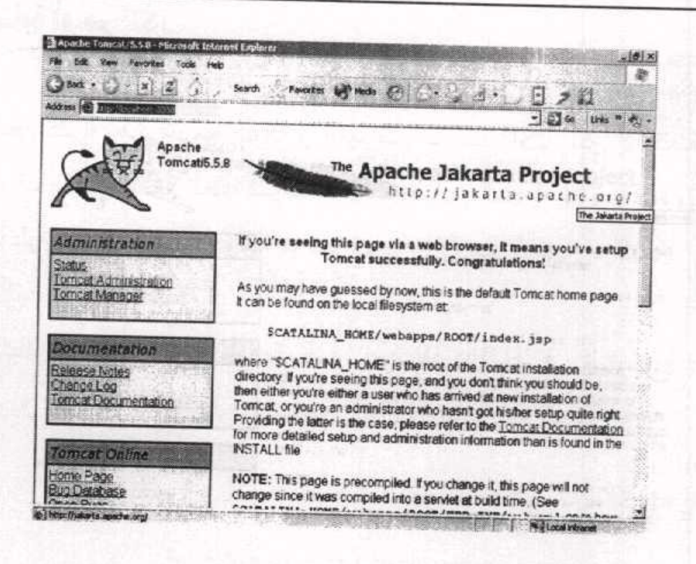


- بعد فترة وجيزة سوف تشاهد شريط تحميل برنامج الخدمة الخاص بـ Tomcat ويجب أن تلاحظ أيقونة صغيرة في شريط الحالة System tray لريشة بها مفتاح أخضر تدل على تشغيل برنامج المترجم.
- إذا كان برنامج المزود لم يتم تحميله أختار الأيقونة Monitor Tomcat من قائمة Start\Programs\Apache Tomcat 5.5 ستجد أيقونة لريشة في أسفل يمين الويندوز (بجانب الساعة) وهي تسمى منطقة تحميل البرامج المخبئة System Tray .

فإذا كانت توجد رسمة مربع أحمر صغير على الريشة هذا معناه أن برنامج الخدمة Service لم يتم تحميله بعد ، فقم بالضغط على المفتاح الأيمن من الماوس بعد التأشير على الأيقونة واختار الأمر Start Service فيتم تحميل المزود كما يتضح من الشكل.



- للتأكد النهائي من عمل مزود الخدمة Tomcat قم بفتح برنامج Internet Explorer وقم بكتابة العنوان الآتي في سطر العنوان: http://localhost:8080
- فإذا لم تظهر الصفحة الخاصة بالمعلومات عن المزود فقد قمت بعمل خطأ ما ، قم بمراجعة خطوات التركيب مرة أخرى.



بهذا تكون مستعدا لاختبار أمثلة هذا الكتاب .

كتابة أوامر JSP

تعتمد هذه اللغة على قواعد لغة الجافا الشهيرة Java لذلك سيكون أسهل كثيرا على مبرمج الجافا أن يفهم بسرعة لغة JSP ولكن إذا كنت لا تعرف شيئا عن لغة الجافا فسوف تستطيع بمنتهى السهولة بناء برامج ويب تفاعلية بسرعة وبسهولة وهذا ما تميزت به JSP منذ الإصدارة الثانية.

مثال:

يمكنك بمنتهى السهولة طباعة كلمة "Hello World" في المتصفح عن طريق تضمين كود JSP في صفحة HTML عادية كما يلي:

```
<html>
<body>
<%
out.println("<h1>Hello World</h1>");
%>
</body>
</html>
```

قم الأن بكتابة الكود السابق في أي محرر نصوص ويمكنك استخدام محرر Macromedia Dreamweaver ولكننا سنلتزم في هذا الكتاب باستخدام المحرر المجاني Notepad الذي يأتي مع جميع إصدارات الويندوز .

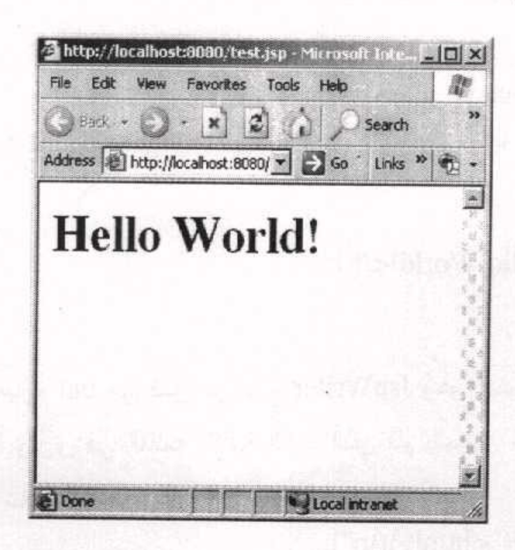
قم بحفظ الملف بالاسم test.jsp في الدليل الفرعي

C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\ROOT\

وهذا إذا كان برنامج Tomcat تم تركيبه على المشغل C . قم الآن بفتح برنامج Internet Explorer وقم بكتابة العنوان الآتي في سطر العنوان

http://localhost:8080/test.jsp

وستكون النتيجة كما بالشكل:



لاحظ أن كلمة "Hello World" قد تم طباعتها بخط كبير كالعناوين وهذا تأثير استخدام وسم HTML الخاص بالعناوين <H1> فإذا لم تكن تعرف هذا الوسم جيدا يجب عليك معرفة نصوص HTML أولا .

يقوم المتصفح IE بعمل طلب Request إلى مزود الويب الذي هو في حالتنا هو المزود Tomcat والذي بدوره يقوم بنقل الطلب برمته إلى كائن يسمى JSP Container وفي أول مرة يقوم فيها بتنفيذ الطلب يتم ترجمة النص السابق كتابته إلى كود بالكامل ثم يتحول إلى ملف تنفيذي يسمى servlet وتأخذ هذه العملية في أول مرة بعض الوقت نظرا للوقت الذي تتم فيه الترجمة ، لذلك حتى يقلل محتوى JSP Container عدد المرات اللازمة للترجمة فإنه يقوم بمقارنة آخر تاريخ تعديل لـــ servlet فإذا كان أقدم لا يقوم المحتوى بإعادة الترجمة مرة أخرى .

إذا أردت رؤية الكود الناتج من عملية الترجمة يمكنك ذلك عن طريق الأوامر :

View Source

فتجد أن الكود تم تحويله إلى نص HTML على الصورة التالية

```
<html>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

لاحظ أن العبارة out هي كائن من نوع JspWriter وهو يستخدم لطباعة النصوص ، ونرى في القائمة التالية الكود الذي قام بتحويل النص الذي قمنا بكتابته سابقا إلى نص HTML :

```
out.write("<html>\r\n");
out.write("<body>\r\n");
out.println("<h1>Hello World!</h1>");
out.write("\r\n");
out.write("</body>\r\n");
out.write("</html>\r\n");
```

ويختلف مكان الملف المصدر الخاص بـــ servlet من مزود لآخر بالنسبة للمزود Tomcat ستجده بعد الدليل tomcat في المسار:

work\Catalina\localhost_\org\apache\jsp

وقد يكون من المفيد أحيانا النظر إلى الملف المصدر في حالة تتبع الأخطاء أثناء التنفيذ .

★ استخدام الوسمين <% %>

كما رأينا سابقا يتم استخدام الوسمين <% %> ليدل على أن النص الموجود بداخلهم هو نص لغة Java ويمكننا توضيح ذلك عن طريق المثال الآتي:

```
<%@ page contentType="text/html; charset=windows-1256" %>
<html dir="rtl">
<body>
التوقيت الآن
<%
java.util.Calendar currTime = new
java.util.GregorianCalendar();
 if (currTime.get(currTime.HOUR_OF_DAY) < 12)
  {
%>
    صباحا
<%
  }
  else if (currTime.get(currTime.HOUR_OF_DAY) < 18)
  {
%>
    ظهرا
<%
  }
  else
  {
%>
    مساء
<%
  }
%>
</body>
</html>
```

نلاحظ من الكود السابق ما يلي:

- يقوم هذا البرنامج بمعرفة الوقت الحالي من اليوم ويقوم بطباعة الكلمة المناسبة للتوقيت وتم ذلك بكتابة نص JSP داخل الوسمين <% %> حتى يتم ترجمة النص إلى كود Java ولاحظ أن العبارات التي خارج الوسم يتم طباعتها كما هي فيتم طباعة كلمة "التوقيت الآن" كما هي.
- قمنا باستخدام المتغير currTime من نوع java.util.Calendar الذي يمكن عن طريقه الوصول إلى الوقت والتاريخ الحالي بجهاز الكمبيوتر وتم إنشاء الكائن عن طريق العبارة new ولكن ليس مجالنا الآن شرح الكائنات.
- من المثال السابق تم استخدام العبارة الشرطية if لتحديد الوقت الحالي بنظام 24 ساعة وتكون النتيجة هي طباعة الكلمة المناسبة .
- لاحظ أن أول وثاني سطر هما المسئولان عن أظهار حروف اللغة العربية بطريقة ملائمة من اليمين لليسار.

```
<%@ page contentType="text/html; charset=windows-1256"
%>
<html dir="rtl">
```

- لا تتم طباعة الكلمات الثلاث صباحا وظهرا ومساء مع بعض ولكن يتم طباعة الكلمة المناسبة فقط نتيجة للكود المكتوب .

وفيما يلي نرى الكود الناتج من عملية الترجمة :

```
out.write("<html>\r\n");
    out.write("<body>\r\nGood\r\n");
   java.util.Calendar currTime = new
java.util.GregorianCalendar();
   if (currTime.get(currTime.HOUR_OF_DAY) < 12)
   {
```

```
  out.write("\r\n       Morning!\r\n");
 }
 else if (currTime.get(currTime.HOUR_OF_DAY) < 18)
 {
  out.write("\r\n       Afternoon!\r\n");
 }
 else
 {
  out.write("\r\n       Evening!\r\n");
 }
  out.write("\r\n");
  out.write("</body>\r\n");
  out.write("</html>\r\n");
```

يمكنك الآن إدراك كيف يقوم الوسمين <% %> بتضمين النص بينهم وأن أي نص خارج الوسمين يتم استبداله بالعبارة out. write التي تقوم بطباعة النص فقط .

★ **استخدام الوسمين <%= %> لطباعة قيمة**

يمكننا كتابة الكود السابق بطريقه أخرى محكمة مع تأدية نفس الوظيفة عن طريق استخدام الوسمين <%= %> لطباعة محتويات متغير مثلا كما يلي:

```
<%@ page contentTypc="text/html; charset=windows-
1256" %>
<html dir="rtl">
<body>
<%
  java.util.Calendar currTime = new
java.util.GregorianCalendar();
  String timeOfDay = "";
```

```
  if (currTime.get(currTime.HOUR_OF_DAY) < 12)
  {
    timeOfDay = "صباحا";
  }
  else if (currTime.get(currTime.HOUR_OF_DAY) < 18)
  {
    timeOfDay = "ظهرا";
  }
  else
  {
    timeOfDay = "مساء";
  }
%>
 التوقيت الأن <% out.write(timeOfDay); %>
</body>
</html>
```

لاحظ طباعة المتغير عن طريق الكائن out ويجب أن تنتهي العبارة بفاصلة منقوطة لأنها عبارة عن سطر كود بلغة الجافا ولكن بدلا من طباعة كل المتغيرات عن طريق استخدام الكائن out تمنحنا لغة JSP الوسمين السابقين لعرض قيم المتغيرات المختلفة فيمكننا كتابة السطر الخاص بالطباعة كما يلي:

```
التوقيت الآن <%= timeOfDay %>
```

ولا يجب كتابة فاصلة منقوطة في هذه الطريقة ويمكننا كتابة أي تعبير منطقي نريد أن نرى نتيجته داخل الوسمين السابقين فمثلا يمكننا كتابة التعبير الحسابي التالي:

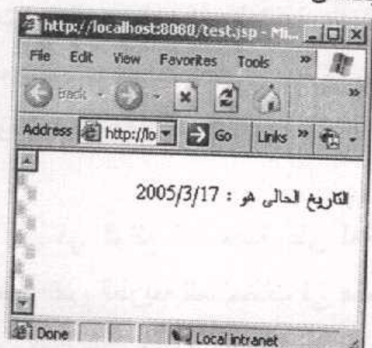
```
<%= 2+5*7-1*8 %>
```

مثال:

في المثال التالي يمكننا أن نرى استخدام الوسمين بطريقة أخرى لعرض تاريخ اليوم:

```
<%@ page contentType="text/html; charset=windows-1256" %>
<html dir="rtl">
<body>
<% java.util.Calendar currDate = new java.util.GregorianCalendar();
  int month = currDate.get(currDate.MONTH)+1;
  int day = currDate.get(currDate.DAY_OF_MONTH);
  int year = currDate.get(currDate.YEAR);
%>
التاريخ الحالي هو: <%= day %>/<%= month %>/<%= year %>
</body>
</html>
```

وتكون النتيجة كما بالشكل:



وتمكننا هذه الطريقة من تطوير موقع بلغة JSP عن طريق عدة أشخاص فيقوم المطورين بتخزين كل المعلومات المطلوبة في الصفحة ثم يقوم المصممين باستخدام الوسمين <%= %> في المكان الملائم للتصميم لعرض المعلومات.

★ إدراج الملاحظات

تعتبر الملاحظات من العناصر الأساسية لأي لغة برمجة وتستخدم لإيضاح فائدة جزء معين من الكود بحيث يمكنك تذكر ماذا تكون نتيجة تنفيذ هذه السطور من الكود بعد مراجعة البرنامج بعد فترة طويلة والاستخدام الآخر للملاحظات هو إيقاف مؤقت لسطر أو عدة سطور بحيث يتجاهلها المترجم عند تنفيذ البرنامج أو عرض الصفحة ، يمكنك أيضا استخدام طريقة نصوص HTML للملاحظات وهي دمج الملاحظات بين الوسمين <!-- -->.

مثال:

<!--ملاحظات على طريقة HTML -->

وتعتبر هذه الطريقة مفيدة عندما تقوم بعملية تنقيح البرنامج أو debugging بحيث يمكنك معرفة مكان المشكلة عند فتح ملف المصدر الخاص بالصفحة دون أن يرى المستخدم هذه الملاحظات عند عرض الصفحة ولكن لا تقوم بوضع بيانات سرية في الملاحظات مثل كلمة سر مستخدم مثلا.

وبما أن لغة JSP هي في الواقع لغة مبنية على لغة الجافا JAVA الشهيرة فيمكنك استخدام نفس الطريقة للملاحظات في هذه اللغة . فمثلا إذا أردت كتابة ملاحظات في سطر واحد أو إيقاف تنفيذ سطر واحد ضع قبله العلامتين //

مثال:

<% // سطر ملاحظات %>

ولا تنس كتابة الكود بالطبع بين وسمي JSP <% %>

وهناك طريقة أخرى لإدراج الملاحظات وهي استخدام العلامتين /* */ ويمكن استخدامهم لكتابة عدة أسطر من الملاحظات أو إيقاف عدة أسطر من الكود عن التنفيذ.

مثال:

```
<% /* %>
   سطر ملاحظات
<% */ %>
```

هناك عيب لهذه الطريقة وهي أنها مربكة عند المتابعة وأيضا يقوم المترجم بتحويل الملاحظات إلى نص فيمكن رؤيتهم عند رؤية ملف المصدر لذلك هناك طريقة أفضل لإدراج الملاحظات وهي استعمال العلامتين <%-- --%> وهذه الطريقة تمنع ظهور الملاحظات في الملف المصدر ويتجاهل المترجم تماما هذه السطور.

مثال:

```
<%-- ملاحظات لن تظهر في ملف المصدر --%>
```

مثال:

إذا أردت إيقاف عدة سطور يمكنك استخدام الطريقة التالية:

```
<%--
<%
  out.println ( "هذا السطر لن يتم تنفيذه أبدا" ) ;
%>
--%>
```

★ الإعلان عن المتغيرات والإجراءات:

يتم استخدام المتغيرات variables لتخزين قيمة معينة بها بحيث يمكن أجراء عمليات على هذا المتغير فيما بعد أو عرض قيمته كما رأينا مع الوسمين <%= %>.

أما الإجراءات methods فهي عبارة عن مجموعة سطور من الكود التي تؤدى وظيفة متكررة فمثلا إذا كنا دائما نحتاج أن نقوم بحساب ضريبة معينة بناء على قيمة يقوم المستخدم بإدخالها فنقوم بإنشاء هذا الإجراء بحيث يمكن بمجرد كتابة اسمه ضمن كود الصفحة أن يقوم بحساب الضريبة ويرجع قيمتها.

مثال:

حتى نستطيع أن نقوم بالإعلان عن المتغير أو الإجراء يجب أن يتم ذلك بين الوسمين <%! %> كما يلي:

```
<%@ page contentType="text/html; charset=windows-1256" %>
<html dir="rtl">
<body>
<%!
  public String myMethod(String anyParameter)
  {
          return "لقد قمت بإدخال القيمة :  "+anyParameter;
  }
%>
<%= myMethod ( "عالم الجافا" ) %>
</body>
</html>
```



كما ترى بمجرد كتابة أسم الإجراء my Method وإدخال القيمة "عالم الجافا" كمعامل حرفي تم تنفيذ الإجراء وإظهار قيمة هذا المعامل . بالطبع هذا المثال للتوضيح فقط ويجب أن تتم العمليات على المعامل قبل إظهار قيمته.

بالمثل يمكنك الإعلان عن المتغيرات بنفس الطريقة بحيث يمكن الوصول إلى هذا المتغير من داخل الإجراء الذي قمت بتعريفه بالوسمين !%< %> أومن كود الجافا ضمن الوسمين %< %> كما يلي

```
<%! int myVar = 22; %>
```

السطر السابق يقوم بالإعلان عن متغير ليحمل عدد صحيح integer اسمه myVar وأعطيناه القيمة 22 ويمكن في أي مكان بالكود الوصول إلى هذا المتغير كما يلي:

```
<%
    out.println ( " =قيمة المتغير  "+ myVar);
%>
```

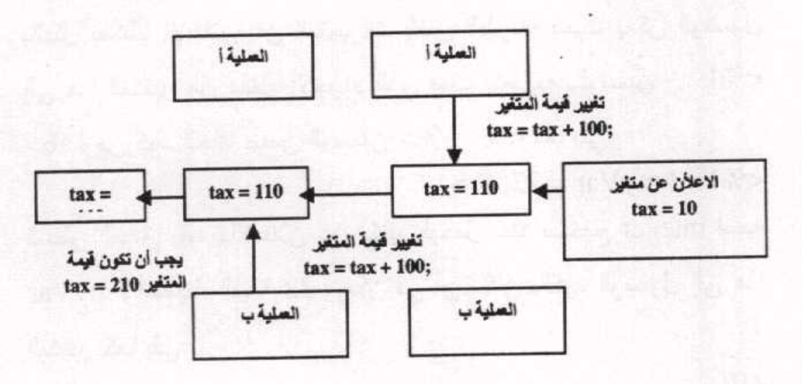
بالمثل يمكنك أظهار قيمة متغير باستخدام الوسمين =%< %> هكذا:

```
=قيمة المتغير  <%= myVar %>
```

✶ التعامل مع عمليات الذاكرة Threads:

مصطلح thread يعبر عموما عن إمكانية تنفيذ نفس البرنامج أو عرض نفس الصفحة عدة مرات في نفس الوقت بدون أن يحدث تداخل بينهم بحيث يكون لكل عملية الذاكرة الخاصة بها وعند إيقاف عملية لا يتم التأثير على إجراء العمليات الأخرى.

كمبرمج جافا يجب أن تفهم كيفية التعامل مع العمليات أو خيوط الذاكرة فإذا كتبت كل الكود الخاص بك داخل الوسمين <% %> فإن برنامجك آمن بالنسبة للتعامل المتعدد أما إذا استخدمت كائنات خارجية أو قمت بالإعلان عن متغيرات خارجية باستخدام الوسمين <%! %> مثلا فيجب أن تراعي هنا أمكانية تنفيذ الكود في نفس الوقت من أكثر من عملية ويمكن توضيح ذلك من الرسم كما يلي:



وتكون المشكلة هي استخدام نفس المتغير لكل العمليات مما يؤدى للتداخل بينهم وتغيير قيمة هذا المتغير.

مثال:

في هذا المثال يتم إدخال عدة قيم مثل الاسم والعنوان ولكن يتم الإعلان عن كل قيمة بمتغير مما يؤدى للتداخل وتغيير قيم هذه المتغيرات إذا تم عرض الصفحة في نفس الوقت من قبل عدة مستخدمين.

```
<html>
<body>
<%!
// الإعلان عن المتغيرات
   String firstName;
   String middleName;
   String lastName;
   String address1;
   String address2;
   String city;
   String state;
   String zip;
%>
<%
// الوصول إلى قيم المتغيرات
firstName = request.getParameter("firstName");
middleName = request.getParameter("middleName");
lastName = request.getParameter("lastName");
address1 = request.getParameter("address1");
address2 = request.getParameter("address2");
city = request.getParameter("city");
state = request.getParameter("state");
zip = request.getParameter("zip");
// Call the formatting routine.
   formatNameAndAddress(out);
```

```
%>
</body>
</html>
<%!
// طباعة الاسم والعنوان
void formatNameAndAddress(JspWriter out)
throws java.io.IOException
{
out.println("<PRE>");
out.print(firstName);
// طباعة أسم الأب إذا كان يوجد به بيانات
if ((middleName != null) && (middleName.length() > 0))
{
out.print(" "+middleName);
}
out.println(" "+lastName);
out.println(address1);
// طباعة العنوان الآخر إذا كان يوجد به بيانات
if ((address2 != null) && (address1.length() > 0))
{
out.println(address2);
}
out.println(city+", "+state+" "+zip);
out.println("</PRE>");
}
%>
```

تسمى هذه الطريقة – لإظهار البيانات – بطريقة النماذج Forms وسنتعلمها بالتفصيل لاحقا ، ويمكن استخدام العبارة synchronized لمنع

تداخل المتغيرات مع بعضها عند التنفيذ ولكن ستؤثر على أداء البرنامج لأن المعالج سيقوم بإنشاء ذاكرة خاصة لكل طلب للمتغير .

```
  synchronized (this)
  {
    firstName = request.getParameter("firstName");
    middleName = request.getParameter("middleName");
    lastName = request.getParameter("lastName");
    address1 = request.getParameter("address1");
    address2 = request.getParameter("address2");
    city = request.getParameter("city");
    state = request.getParameter("state");
    zip = request.getParameter("zip");
// النداء على أجراء أظهار البيانات
    formatNameAndAddress(out);
  }
```

لذلك يجب استخدام الإجراءات وإعطاء الإجراء القيم المختلفة كمعامل حتى يتم تنفيذ الإجراء في ذاكرة خاصة به كما يلي:

```
<html>
<body>
<%
// الإعلان عن كائن ليحمل المتغيرات المختلفة
  NameAndAddress data = new NameAndAddress();
// الوصول إلى قيم المتغيرات
  data.firstName = request.getParameter("firstName");
  data.middleName =
request.getParameter("middleName");
  data.lastName = request.getParameter("lastName");
  data.address1 = request.getParameter("address1");
  data.address2 = request.getParameter("address2");
```

```
  data.city = request.getParameter("city");
  data.state = request.getParameter("state");
  data.zip = request.getParameter("zip");
// النداء على إجراء إظهار البيانات
  formatNameAndAddress(data, out);
%>
</body>
</html>
<%!
// الكائن الذي يحمل البيانات المختلفة
  class NameAndAddress
  {
    public String firstName;
    public String middleName;
    public String lastName;
    public String address1;
    public String address2;
    public String city;
    public String state;
    public String zip;
  }
// طباعة الاسم والعنوان
  Void formatNameAndAddress(NameAndAddress  data,
JspWriter out)
    throws java.io.IOException
  {
    out.println("<PRE>");
    out.print(data.firstName);
// طباعة الاسم الأوسط إذا كان به بيانات
    if ((data.middleName != null) &&
```

```
        (data.middleName.length() > 0))
      {
        out.print(" "+data.middleName);
      }
      out.println(" "+data.lastName);
      out.println(data.address1);
// طباعة العنوان إذا كان به بيانات
      if ((data.address2 != null) &&
        (data.address1.length() > 0))
      {
        out.println(data.address2);
      }
      out.println(data.city+", "+data.state+" "+data.zip);
      out.println("</PRE>");
    }
%>
```

يتم في هذا المثال تفادى التصادم الذي يمكن أن يحدث بين قيم المتغيرات عن طريق إنشاء كائن ليحمل المتغيرات المتعددة ولأن كل عملية تقوم باستخدام نسخة منفصلة من هذا الكائن فلا يتم تغيير قيم هذه المتغيرات.

```
if ((dam. [illegible]
  (dew.se [illegible]
  }
  out.println( [illegible]
}
out.println(dam.city+ [illegible] "+dam.zip);
out.println("</PRE>");
}
```

[illegible]

الفصل الثاني

التعامل مع SERVLETS

كما تعلمنا سابقا فإن Servelt هي المحتوى الأساسي الذي يتم ترجمة نص JSP إليه وتعلم شيئا عن هذا المحتوى مفيد جدا حتى يمكنك أن تعرف ماذا يجرى بالفعل عند كتابة نصوص JSP ، كما أن لها فائدة كبيرة عند مرحلة تنقيح البرنامج ومعرفة سبب المشكلة ، وعادة عندما تتقدم في لغة JSP سوف تقوم باستخدامها جنبا إلى جنب مع السيرفلت.

وتمكنك السيرفلت أيضا من التعامل مباشرة مع طلبات النماذج المرسلة إلى مزود الخدمة وأيضا تشكيل الاستجابة المناسبة لهذا الطلب وبعكس نصوص JSP التي تهتم بطريقة عرض البيانات واستخدامها ضمن نصوص HTML فإن السيرفلت تعتمد على لغة الجافا مما يعطيك قوة كبيرة لكتابة كود دوال يمكنها التعامل مع البنية التحتية للبيانات أو وضع قواعد لنظام العمل ببرنامجك.

مثال:

يمكننا تنفيذ نفس المثال الخاص بعرض كلمة الترحيب الشهيرة "Hello World" باستخدام السيرفلت.

- قم الآن بفتح محرر النصوص Notepad وقم بكتابة الكود الآتي ثم قم بحفظه بالاسم HelloWorldServlet . java

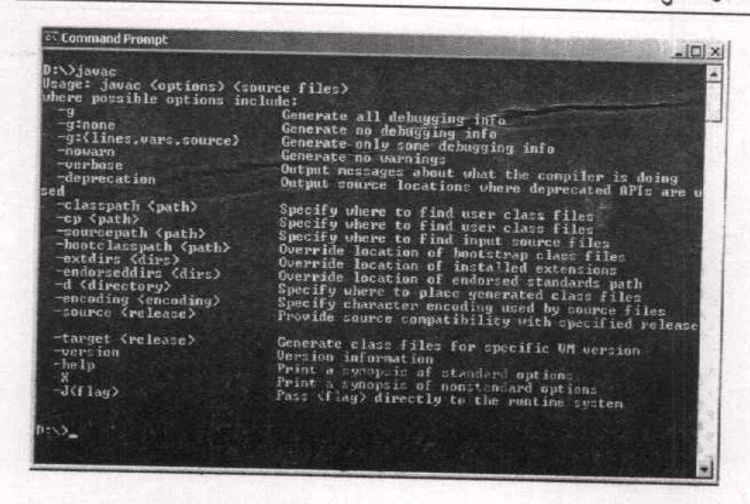
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class HelloWorldServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
            HttpServletResponse response)
    throws IOException
  {
```

```
// لأعلام مزود الويب أن نوع الاستجابة هي نصوص HTML
    response.setContentType("text/html");
// الوصول إلى كائن الطباعة out
    PrintWriter out = response.getWriter();
//كتابة نصوص HTML فى المتصفح
    out.println("<html>");
    out.println("<body>");
    out.println("<h1>Hello World!</h1>");
    out.println("</body>");
    out.println("</html>");
  }
}
```

- ميزة الكتابة مباشرة باستخدام السيرفلت أنك لا تحتاج أن تتعلم قواعد جديدة للغة ، فقط قم بإتباع نفس الخطوات في المثال السابق ولكن العيب الوحيد أنك سوف تقوم بعمل يدوي أكثر من إتباع طريقة JSP ولكن كما ترى فإن العمل اليدوي ليس شيئا كبيرا .
- حتى نقوم بتنفيذ الكود السابق يجب عليك معالجة الكود ليتحول إلى ملف تنفيذي بالامتداد class وحتى يتم ذلك سوف نقوم باستخدام مترجم الجافا الملف javac.exe الذي يأتي مع تركيب مكتبة الجافا JDK كما رأينا في المقدمة ، وهذا البرنامج من البرامج التي تعمل في بيئة نظام التشغيل دوس DOS ويجب كتابته بالصيغة الآتية :

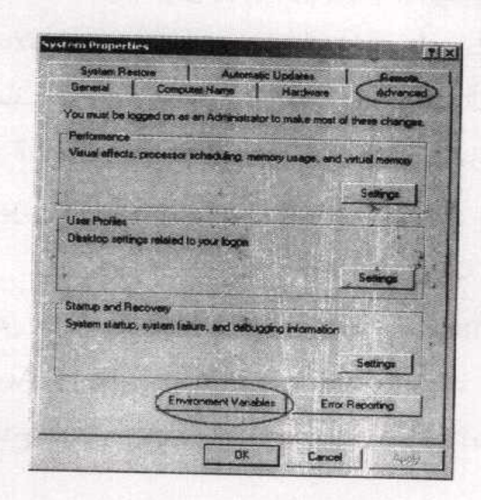
javac <اختيارات التنفيذ> ملف الجافا

ويتم ذلك عن طريق فتح نافذة الدوس من قائمة Start | Programs | Accessories واختيار Command Prompt أكتب الأمر javac واضغط مفتاح الإدخال Enter وشاهد التعليمات المساعدة لهذا الأمر .

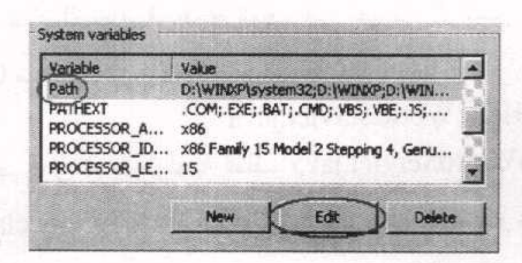


إذا لم يتم تنفيذ الأمر وظهرت رسالة خطأ فإن الملف ليس موضوع في المسار الافتراضي **Path** ويجب أن يتم وضعه كما يلي:

- قم بالضغط بالمفتاح الأيمن من الماوس على أيقونة My Computer واختار Properties ثم أختار الصفحة Advanced ثم أضغط على المفتاح Environment Variables كما بالشكل:

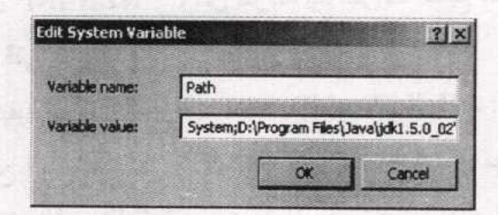


- سوف تظهر نافذة بها جميع متغيرات النظام ومتغيرات المستخدم الحالي قم الآن باختيار المتغير Path في المجموعة System Variables ثم اضغط على المفتاح Edit:



- تظهر نافذة صغيرة بها المسار الافتراضي لبعض الأدلة الهامة قم الآن بإضافة مسار مكتبة الجافا JDK إلى هذا السطر ولا تنس إضافة فاصلة منقوطة للفصل بين المسار والأخر ، فمثلا إذا تم تركيب المكتبة في المشغل D قم بكتابة

;D:\Program Files\Java\jdk1.5.0_02\bin



- بعد إضافة المسار قم بالضغط على مفتاح Ok ثم Ok لنافذة المتغيرات ثم Ok لنافذة الخواص Properties بهذا يصبح الأمر javac في المسار الافتراضي ويمكن تشغيله من أي مكان.

- يجب الآن ترجمة Compile الملف HelloWorldServlet.java وحتى يتم ذلك يجب كتابة المسار الذي يوجد به الملف ثم اسم الملف ولكن نظرا لأننا نكون في بيئة DOS فإن بعض الأسماء الكبيرة تسبب مشكلة ، لذلك للتسهيل قم بنسخ الملف إلى الفهرس الرئيسي لأي مشغل وليكن C مثلا ثم اكتب الأمر الأتي ثم أضغط Enter:

```
javac c:\HelloWorldServlet.java
```

يأخذ المترجم ثوان ثم يتم ترجمة الملف HelloWorldServlet.java إلى الملف HelloWorldServlet.class والذي يمكن تنفيذه وعرضه في المتصفح.

- قم الآن بنسخ الملف HelloWorldServlet.class إلى الدليل WEB-INF\classes المتفرع من الدليل الرئيسي ROOT لبرنامج المزود Tomcat لأن هذا المسار خاص بتنفيذ صفحات السيرفلت .
- حتى يمكننا تنفيذ السيرفلت يجب تعديل ملف اسمه web.xml – سنجده في الدليل WEB-INF – وعن طريقه يتم تعريف جميع صفحات الجافا سيرفلت التي يتم تنفيذها .

وفيما يلي كيفية تعريف السيرفلت HelloWorldServlet به:

1. أفتح الملف عن طريق الضغط على المفتاح الأيمن من الماوس ثم أختار الأمر Edit الذي يفتح الملف في نافذة برنامج Notepad .
2. قم بالبحث عن الجزء الذي يوجد به التعليق <!-- JSPC servlet mappings start --> ثم اكتب بعده مباشرة وسمين لتحديد أسم السيرفلت كما يلي:

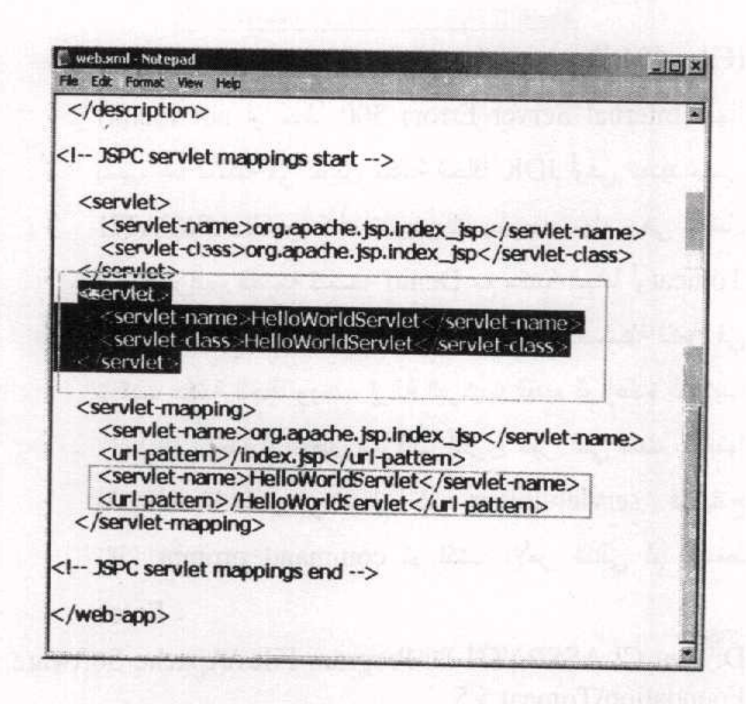
```
<servlet>
   <servlet-name>HelloWorldServlet</servlet-name>
```

```
    <servlet-class>HelloWorldServlet</servlet-class>
  </servlet>
```

3. ثم قم بإضافة الوسمين الآتيين بعد الفقرة <servlet-mapping> لتحديد العنوان المستخدم للنداء على السيرفلت كما يلي

```
<servlet-name>HelloWorldServlet</servlet-name>
<url-pattern>/HelloWorldServlet</url-pattern>
```

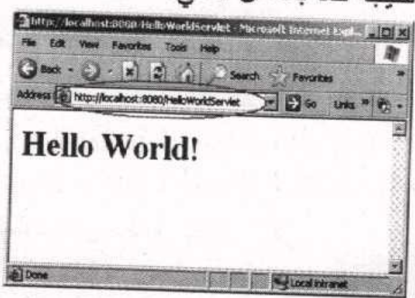
ويجب أن يكون الملف الآن مطابق للشكل الآتي:



4. أغلق الملف بعد حفظه ثم قم بفتح مستعرض الإنترنت واكتب السطر الآتي في شريط العنوان:

http://localhost:8080/HelloWorldServlet

ويجب أن تكون النتيجة كما بالشكل التالي:



5. إذا لم تظهر الصفحة وظهرت بدلا منها صفحة خطأ 404 (File not found) أو خطأ 500 (Internal Server Error) فهذا بسبب إما مشكلة في تنزيل مكتبة الجافا JDK أوفي تحديد مسار CLASSPATH ويمكنك تحديد ذلك بطريقه سهله وهى تشغيل أحد السيرفلت القادمة كنسخه Demo مع مكتبة الجافا أو Tomcat ومحاولة عرضها فإذا لم يتم عرضها أيضا فإن المشكلة تكمن في تركيب مكتبة الجافا ويجب إزالة التركيب القديم ثم إعادة التركيب ، أما إذا استطعت تشغيل سيرفلت أخرى غير التي قمت بكتابتها فالمشكلة هنا تكمن في مسار ملف servlet-api.jar قم بفتح نافذة command prompt ثم اكتب الأمر التالي ثم أضغط Enter :

```
D:\>set CLASSPATH D:\Program Files\Apache Software
Foundation\Tomcat 5.5
\common\lib\servlet-api.jar;D:\Program Files\Apache
Software
Foundation\Tomcat 5.5\webapps\servlets-examples\WEB-
INF\classes\;%CLASSPATH%
```

هذا بفرض أن برنامج Tomcat تم تركيبه في المشغل D:\

وقد قمنا بعمل هذه الخطوة لأن برنامج Tomcat يحتاج إلى الأدوات (classes) الموجودة في الملف servlet-api.jar ويوجد هذا الملف في المسار common\lib وقديما كان يسمى servlet.jar .

لاحظ أن برنامج مزود الويب Tomacat 5 هو أيضا يأتي كجزء من تركيب مكتبة الجافا JDK وهذا لأن برنامج Tomcat يعتبر المزود القياسي لتنفيذ صفحات JSP ذات الإصدار 2 والسيرفلت حتى الإصدار 2.4 ويتم دائما تطوير هذا البرنامج واختباره مع صفحات JSP.

- **شرح كود المثال:**

هناك طريقتين لإنشاء سيرفلت: **الأولى هي عن طريق توريث كائن لكائن أخر مزود بتعريف لواجهة الكائن الأساسي** servlet **والثانية هي تعريف واجهة الكائن الأساسي** servlet **مباشرة في السيرفلت ، ففي** المثال السابق Servlet World Hello قمنا باستخدام الطريقة الأسهل واستخدمنا كائن مورث هوالكائن Servlet Http .

عندما يتم أنشاء طلب request لسيرفلت معينه فإن مكتبة الجافا تقوم بتحميل السيرفلت في الذاكرة إذا لم يتم تحميله من قبل (أول طلب) وتقوم باستدعاء الوسيلة service الخاصة بالسيرفلت وتأخذ هذه الوسيلة معاملين: **الأول** هو كائن يحتوى على معلومات عن الطلب من المستعرض **والثاني** كائن يحتوى على معلومات عن الاستجابة للمستعرض والكائن Servlet Http يحتوى بالفعل على تعريف تلقائي لهذه الوسيلة حيث يقوم بالنظر إلى رأس حالة الطلب في عنوان HTTP ثم يحدد أنها الوسيلة GET فيقوم باستدعاء الوسيلة do Get وهى التي قمنا بتعديلها وكتابة الكود الذي يخصنا بها .

وسوف نقدم فيما بعد في فصل التعامل مع النماذج طرق أخرى للتعامل مع الطلب . وبالنسبة للوسيلة do Get فإنها تخص الكائن Servlet Http لمعالجة الطريقة GET بينما تستطيع الفئة (class) Servlet Generic من استخدام الوسيلة service لمعالجة جميع أنواع حالة الطلب (request).

بعد ذلك كما بالمثال يجب أن تخبر السيرفلت نوع حالة الاستجابة وفى معظم الحالات يقوم بالاستجابة بنصوص من نوع HTML لذلك نحدد النوع هنا على أنه *text/html* وهذا يختلف إذا قمنا بالاستجابة عن طريق لغة البيانات المعرفة XML فنحدد هنا النوع *text/xml* حتى يتم إرجاع البيانات في صيغة ملائمة لصيغة XML . وقد قمنا في المثال السابق بتحديد نوع الاستجابة عن طريق الوسيلة set Type Content وهى أحد وسائل الكائن response الذي تم إرجاعه لنا كمعامل من معاملي Get do

.

بعد ذلك تكون مستعدا لترسل بيانات التي تريدها إلى المستعرض ، ولذلك سنحتاج إلى كائن يقوم بهذه العملية . ومرة أخرى يحتوى الكائن response على الوسائل التي تعطى إمكانية طبع الرسائل ، ولأن في هذا المثال نقوم فقط بطبع رسائل نصية فسوف نحتاج إلى الكائن Print Writer فقط ويتم تخصيص إمكانية الطباعة للمتغير out عن طريق الوسيلة get Writer للكائن response أما إذا أردت إرسال بيانات ثنائية (نصوص ورسومات) قم باستخدام الوسيلة get Stream Output

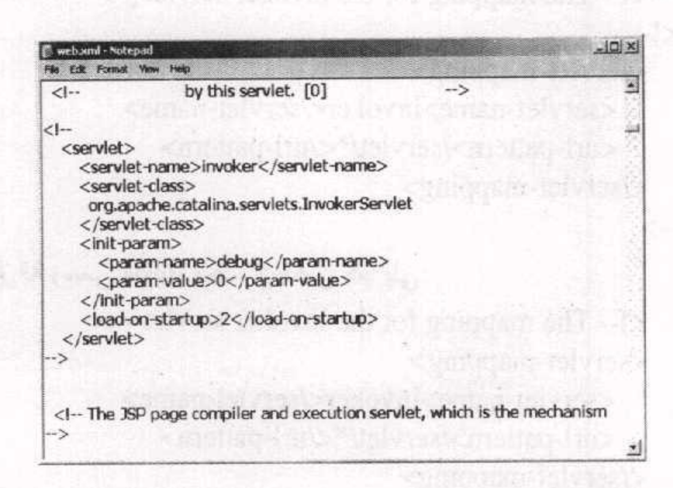
إذا لم تكن لاحظت أننا نقوم بتعريف المتغيرات عن طريق الصيغة الآتية:

[قيمة المتغير] = [أسم المتغير] <نوع المتغير>

وأخيرا قمنا باستخدام الوسيلة المعروفة println من وسائل الكائن Writer Print لإرسال رسائل نصية من نوع HTML إلى المستعرض ، وقد عرفنا فيما سبق كيف أن صفحات JSP تتحول بعد ذلك إلى سيرفلت ، لذلك نلاحظ وجه الشبه بين الاثنين في عملية الاستجابة.

استخدام invoker لاستدعاء السيرفلت

كطريقة أخرى سهلة لتنفيذ السيرفلت يمكنك استخدام السيرفلت invoker والذي يأتي مع مكتبة الجافا ويمكن التعامل معه بطريقة مباشرة ولكن يتم إغلاق هذه الإمكانية في برنامج Tomcat لدواعي أمنية حيث تسمح هذه السيرفلت باستدعاء وتنفيذ أي سيرفلت على جهاز الكمبيوتر الخاص بك بدون الحاجة إلى أي صلاحيات ، وحتى يمكنك تجربة هذه السيرفلت ستحتاج إلى تعديل ملف web.xml الذي يوجد بالدليل /conf وعند فتح الملف يجب أن تبحث عن أسم السيرفلت invoker حتى تصل للوسم الخاص بها كما بالشكل:



```
web.xml - Notepad
File Edit Format View Help
<!--            by this servlet. [0]            -->

<!--
  <servlet>
    <servlet-name>invoker</servlet-name>
    <servlet-class>
    org.apache.catalina.servlets.InvokerServlet
    </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
-->

<!-- The JSP page compiler and execution servlet, which is the mechanism
-->
```

وإذا خمنت فإن invoker محصورة بين وسمى التعليق <!-- --> ويجب إزالة العلامتين حتى تصبح السيرفلت هكذا:

```
web.xml - Notepad
File Edit Format View Help
<!--                by this servlet. [0]                -->

  <servlet>
    <servlet-name>invoker</servlet-name>
    <servlet-class>
     org.apache.catalina.servlets.InvokerServlet
    </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>


 <!-- The JSP page compiler and execution servlet, which is the mechanism
-->
 <!-- used by Tomcat to support JSP pages. Traditionally, this servlet  -->
 <!-- is mapped to the URL pattern "*.jsp". This servlet supports the   -->
```

ويجب أيضا إزالة وسم التعليق في الفقرة الخاصة بتعريف رابط التنفيذ (mapping) والتي تكون على الشكل الآتي:

```
  <!-- The mapping for the invoker servlet -->
<!--
  <servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
-->
```

وبعد إزالة وسمى التعليق يجب أن تكون كما يلي:

```
  <!-- The mapping for the invoker servlet -->
  <servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
```

وهكذا يمكنك تنفيذ واستعراض السيرفلت الخاصة بك عن طريق رابط يحتوى على المسار servlet/ فعندما يجد invoker هذا المسار فإنه يحاول البحث عن الملف التنفيذي (class.*) في أي مسار متضمن في المتغير CLASSPATH وتنفيذه ، ويمكن تنفيذ المثال السابق عن طريق الرابط الأتي:

http://localhost:8080/servlet/HelloWorldServlet

ولضم مسار الملف HelloWorldServlet.class إلى مسار CLASSPATH أستخدم الأمر set CLASSPATH كما سبق.

والميزة في هذه الحالة أننا لا نحتاج إلى تعريف السيرفلت في كل مرة نقوم فيها بتنفيذ سيرفلت جديد للملف web.xml مما يسرع عملية التطوير ولكن على النقيض فكما ذكرنا فإن استخدام السيرفلت invoker يفتح ثغرة أمنية مما يتيح لأي شخص أن يكتب برنامج جافا ثم يتم تحميل هذا البرنامج عن طريق المسار servlet/ لذلك يجب عليك استخدام invoker عند التطوير فقط ولا تضمنه أبدا في كمبيوتر المزود الرئيسي عند الانتهاء من المشروع.

وحتى يمكنك استخدام الملفات التنفيذية (class) في أكثر من برنامج ويب يمكنك وضع هذه الملفات في المسار common\class\ في الدليل الرئيسي لبرنامج Tomcat ويمكنك وضع مكتبات JAR (سنتكلم عنها لاحقا) في الدليل common\lib\ وإذا حاولت تنفيذ سيرفلت أول مرة بعد نقلها إلى مكان جديد ولم يجد Tomcat هذه السيرفلت وظهرت رسالة خطأ فقم بإعادة تمهيد Tomcat وحاول مرة أخرى فإذا ظهرت رسالة خطأ أيضا فأنها مشكلة في مسار المتغير CLASSPATH .

وبنفس الطريقة فإنك أحيانا ستحتاج إلى إعادة تمهيد Tomcat إذا قمت مثلا بتغييرات على السيرفلت التي تم تحميلها بالفعل في الذاكرة وذلك حتى تشاهد هذه التغييرات.

نشر وتنفيذ السيرفلت

في الحياة العملية لا يتم نشر الملف التنفيذي (class) مباشرة على الويب ولكن يتم ذلك عن طريق دمجه في ملف أرشيف مضغوط يسمى WAR اختصار للكلمة (Web Archive) وهذا الملف يمكن أن يتضمن الملفات تنفيذية الخاصة بالبرنامج وملفات نصوص HTML وحتى ملفات تنفيذية أخرى يمكن استدعاؤها مثل الملف WAR والملف JAR ، ولكن الأول يخصص لبرامج الويب .

ويتم تعريف مسار هذه الملفات وكيفية استدعاؤها في الملف web.xml الذي يسمى الملف الوصفي للنشر deployment (descriptor) حيث يحدد بالتفصيل مكونات هيئة الملف WAR وفيما يلي مثال بسيط للملف الوصفي web.xml لنشر السيرفلت : HelloWorldServlet

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>Hello World</display-name>
  <description>مثال على الملف الوصفي</description>
```

```
  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>HelloWorldServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/hi</url-pattern>
  </servlet-mapping>
</web-app>
```

نجد في هذا الملف أننا قد كتبنا الوسمين <? xml?> الذين يعبران على أن الملف نوعه xml طبقا لآخر قواعد محددة في الإصدار 2.4 للغة الجافا ، أما الوسم <web-app> فيحدد نوع البرنامج على أنه برنامج ويب والوسم <display-name> والوسم <description> يحددان وصف مبسط للبرنامج فالوسم <dosplay-name> قد يستخدم لإظهار اسم مختصر للبرنامج عن طريق مزود الويب Tomcat أوفي ملف تتبع التنفيذ (log file) ، أما الوسم <description> فيستخدم لإدراج شرح مطول للبرنامج.

في الملف web.xml يمكن تعريف أي عدد من ملفات السيرفلت التنفيذية ، وكل منها على حده يكون محصورا بين الوسم <servlet> وكما ترى من المثال السابق فإن اسم السيرفلت حدد HelloWorld واسم الملف التنفيذي HelloWorldServlet .

ويجب أيضا تحديد رابط تنفيذ السيرفلت في الوسم <servlet-mapping> والذي يتطلب تعريف وسمين <servlet-name> الذي يحدد اسم السيرفلت السابق الإعلان عنه في الوسم السابق و<url-pattern> الذي

يحدد مسار معرف لتنفيذ السيرفلت في المستعرض ويمكنك تعريف هذا المسار عن طريق القاعدة التالية:

(حالة طلب) = URI(1) مسار بداية البرنامج + (2) مسار السيرفلت (3) +معلومات أخرى(4)

1. الجزء الأول يعبر عن المسار الذي يتم فيه تعريف أسم البرتوكول المستخدم (HTTP/FTP) واسم الكمبيوتر أو رقم IP الخاص ورقم ميناء الاتصال إذا كان رقم آخر غير 80 ثم الاسم المستخدم للوصول إلى السيرفلت ولاحظ هنا أن URI وهى اختصار (Uniform Resource Identifier) تعبر عن الوصول إلى كائن أو مورد معين باسمه بينما المصطلح URL وهو اختصار (Uniform Resource Locator) يعبر عن كيفية الوصول إلى مورد معين ويعتبر URL جزء من URI لأن الأول قد يحتوى على أسم للمورد
2. هو جزء من URI ويفصل بين موارد برنامج وآخر ويعبر عن مسار بداية البرنامج.
3. وهو يعبر عن المسار الذي سيتم البحث فيه عن السيرفلت

وفيما يلي أهم قواعد لهذا الجزء:

- المسار المحصور بين العلامتين "*/" , "/" يستخدمان لتعريف رابط التنفيذ (mapping).
- النص الذي يبدأ بالعلامة "*." يعبر عن امتداد لرابط التنفيذ.
- النص الذي يحتوى فقط العلامة "/" يعبر عن رابط تنفيذ خاص بالسيرفلت الافتراضي للبرنامج.

- أي نص أخر يدرج بعد ذلك يجب أن يطابق أسم السيرفلت تماما .

على سبيل المثال في المثال السابق إذا قمنا بتحديد نموذج المسار كما يلي: "/hiServlet/*"

فإنه يمكننا تنفيذ السيرفلت عن طريق الرابط التالي:

http://localhost:8080/hiServlet/

وإذا قمنا بتحديد نموذج المسار ليشمل النص "*.far" فإنه يمكننا استدعاء السيرفلت عن طريق الرابط التالي:

http://localhost:8080/HelloWorldServlet.far

4. بعد المسار الخاص بالسيرفلت يمكن تحديد أي معلومات أخرى أضافية

إنشاء ملف WAR:

سنقوم الآن بإنشاء الملف WAR الذي سيحتوى على الأدلة التالية

```
WEB-INF/
WEB-INF/web.xml
WEB-INF/classes/
WEB-INF/lib/
```

وكما ذكرنا يمكن تضمين ملفات تنفيذية في المسار WEB-INF/classes وملفات JAR في المسار WEB-INF/lib ويمكن تنفيذ مجموعة السطور القادمة في نافذة command prompt لإنشاء الملف helloworld.war:

```
mkdir WEB-INF
copy web.xml WEB-INF
mkdir WEB-INF\classes
javac WEB-INF\classes HelloWorldServlet.java
jar cvf helloworld.war WEB-INF
```

يجب أن نحترس في كتابة أسم المسار وحتى يتم تنفيذ البرنامج تأكد من كتابة أسم الدليل WEB-INF بحروف كبيرة.

★ نشر ملف helloworld.war

قم بنسخ الملف helloworld.war إلى الدليل webapps ثم قم بإعادة تمهيد برنامج Tomcat (قد لا تحتاج إلى هذه الخطوة) وعندما يعمل البرنامج في المرة الثانية سيقوم بالبحث عن أي ملفات war ثم يقوم بفك هذه الملفات وتكون مستعدة للتنفيذ ، فإذا قمت بأي تعديلات أخرى ثم قمت بنسخ ملف war فإن Tomcat لن يقوم بفك الملفات إذا كانت هناك نسخة سابقة تم تركيبها لذلك يجب عليك أن تحذف يدويا أي تركيب سابق لهذه الملفات .

قم الآن بتنفيذ الخطوات السابقة ولاحظ أنه بعد إعادة تمهيد Tomcat أن هناك دليل يسمى helloworld تم أنشاؤه وهذا هو مسار بداية البرنامج كما ذكرنا سابقا ويكون رابط تنفيذ السيرفلت هو:

http://localhost:8080/helloworld/hi

وهناك الكثير الذي يمكن إدراجه في الملف web.xml لتعديل البرنامج وفقا لاحتياجاتنا سنحاول أن نذكر معظمها لاحقا .

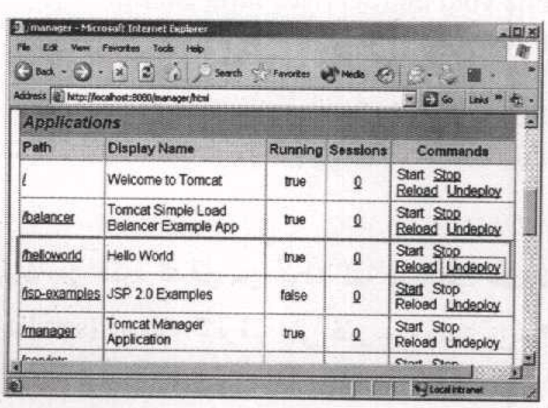
ويمكنك استخدام أداة تسمى Ant لإنشاء ملف war بطريقة تلقائية دون استخدام الأوامر ولكن ستحتاج إلى إنشاء ملف web.xml يدويا في هذه الحالة أيضا. ويمكن تنزيل هذه الأداة من الرابط التالي:

http://jakarta.apache.org/ant

كما يمكنك استخدام صفحة المدير الخاصة ببرنامج Tomcat لتركيب أو إزالة البرامج المثبتة ويمكن الوصول إليها عن طريق الرابط:

http://localhost:8080/

واختيار Tomcat Manager من القائمة ثم قم بإدخال أسم المستخدم admin .



أيضا قد تحتاج إلى تعريف مستخدمين يمكن لهم الدخول إلى هذه الصفحة في الملف tomcat-users.xml الموجود بالدليل conf/ .

البنية الهيكلية للسيرفلت:

في المثال السابق HelloWorldServlet تعلمنا أن السيرفلت هي عبارة عن كائن معرف بلغة الجافا ويتضمن عدة وسائل هامة حتى يمكن التعامل معه ويمكنك الاختيار عند إنشاء السيرفلت أن تقوم بتعريف هذه الوسائل بنفسك أو أن تقوم باستخدام كائن منحدر من سيرفلت موجود من قبل ومعرف به هذه الوسائل. وسنجد في واجهة الكائن العام servlet الوسائل التي تمكننا من التعامل مع السيرفلت . وفيما يلي الكود الخاص بتعريف الكائن العام servlet:

```
package javax.servlet;
public interface Servlet
{
```

```
  public void destroy();
  public ServletConfig getServletConfig();
  public String getServletInfo();
  public void init(ServletConfig config)
    throws ServletException;
  public void service(ServletRequest request,
    ServletResponse response)
    throws ServletException, java.io.IOException;
}
```

والوسائل المدرجة في الكود السابق من أدوات لغة السيرفلت المدرجة مع الجافا (servlet API) ويمكنك التعرف على كثير من وسائل ودوال هذه اللغة عن طريق أحد الرابطين التاليين:

http://jakarta.apache.org/tomcat
http://java.sun.com/j2ee/1.4/docs

ومعظم الوقت ستحتاج عند إنشاء سيرفلت أن تجعله ينحدر إما من الكائن GenericServlet أو الكائن HttpServle ، فكلا من الكائنين يتضمنا واجهة الكائن servlet مع وجود وسائل أضافية تمكنك من تعريف هذه الواجهة ضمن الكود بطريقة سهلة.

* **الوسيلة service:**

يعتبر قلب أي سيرفلت هو الوسيلة service حيث يقوم مزود الويب باستخدام هذه الوسيلة للتعامل مع طلبات المستعرض وبإنشاء السيرفلت من الكائن HttpServlet فنحن لا نحتاج إلى تعريف هذه الوسيلة لأن هذا الكائن قام بذلك لنا ، ولكن يجب أن تعرف أن هذه الوسيلة عند استدعائها تحتوى على كائنين الأول ServletRequest الذي يحتوى على معلومات

عن حالة الطلب الذي قام بتنفيذ السيرفلت والكائن ServletResponse الذي يحتوى على طريقة إرسال الاستجابة .

★ الوسيلة init:

في أحيان كثيرة تحتاج السيرفلت إلى تنفيذ كود تمهيدي مرة واحدة قبل تحميلها ويتم استدعاء هذه الوسيلة بعد تحميل السيرفلت أول مرة ولكن قبل أن تتعامل مع أي حالة طلب.

وفيما يلي مثال يوضح استخدام هذه الوسيلة لفتح اتصال بقاعدة بيانات:

```
protected Connection conn;
public void init()
{
  try
  {
// التأكد من تحميل برنامج تعريف JdbcOdbcDriver قبل استخدامه
 الكائن
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// ODBC محاولة الاتصال بقاعدة البيانات عن طريق
      conn = DriverManager.getConnection(
         "jdbc:odbc:examples");
  }
  catch (Exception exc)
  {
// API في حالة وجود خطأ يتم أظهار معلومات عنها عن طريق دالة
      getServletContext().log(
         "Error making JDBC connection: ", exc);
  }
}
```

لاحظ من الكود السابق أن الوسيلة init لا تأخذ معامل على الإطلاق عند استخدام أحد الكائنين HttpServlet أو GenericServlet أما نفس الوسيلة فهي معرفة في واجهة الكائن servlet بمعامل واحد هوالكائن ServletConfig لهذا فإن الكائن servelt هو المسئول عن الكائن ServletConfig أما الكائنين HttpServlet و GenericServlet فيهتما بالتعامل مع المعامل ServletConfig ثم يقدما تعريف للوسيلة init بدون هذا المعامل لكي تستخدمه للتمهيد فقط.

إذا قمت بتعريف الوسيلة init على أنها تعديل (override) فيجب أن تستخدم في أول سطر العبارة super.init حتى يتم التعامل مع الكائن ServletConfig بطريقه تلقائية

★ الوسيلة destroy:

من قام بدراسة التعامل مع برمجة الكائنات الموجهة (OOP) من قبل يستطيع تخمين ما تفعله هذه الوسيلة ، فعندما يتم إزالة السيرفلت من الذاكرة يتم استدعاء هذه الوسيلة لتنفيذ أي كود يقوم بإنهاء أو تحرير موارد النظام المستخدمة عن طريق السيرفلت وبعكس الوسيلة init فيمكن مثلا التأكد من إغلاق الاتصال مع قاعدة البيانات قبل إنهاء السيرفلت كما يتضح من المثال:

```
public void destroy()
{
  try
  {
// null إغلاق الاتصال فقط إذا كان مفتوح أي لا يساوى
    if (conn != null)
```

```
        {
           conn.close();
        }
     }
     catch (SQLException exc)
     {
// في حالة وجود خطأ يتم إظهار رسالة خطأ
        getServletContext().log(
           "Error closing JDBC connection: ", exc);
     }
}
```

★ الوسيلتان getServletInfo و getServletConfig:

في حالة استخدام أحد الكائنين GenericServlet أو HttpServlet فعادة لن تحتاج إلى تعديل الوسيلتان getServletInfo و getServletConfig فبالنسبة للوسيلة الأولى يمكنك عن طريقها إرجاع نص عادى يحتوى على معلومات عن السيرفلت مثل اسم المطور والإصدار الخاص بها ، أما الوسيلة الثانية فهي تقوم بإرجاع الكائن ServletConfig الذي يتم التعامل معه كمعامل مع الوسيلة init فإذا لم تكن تهتم بالتعامل مع الكائن config فالأفضل أن تجعل الوسيلة init في واجهة الكائن servlet من التعامل معها (هنا الواجهة servlet تسمى الكائن الأساسي superclass).

★ إرسال الاستجابة إلى المستعرض:

هناك اختلاف في حالة الاستجابة بالنسبة للسيرفلت وصفحة JSP فأقل إجراء لعملية الاستجابة بالنسبة للسيرفلت هو تحديد المعامل content

type وكتابة إجراء طبع الاستجابة باستخدام الوسيلة print ، وعن طريق الوسيلة service يتم الوصول إلى كائنين يتم الاستعانة بهما في عملية الاستجابة هما ServletRequest و ServletResponse .

ويختلف المعامل content type باختلاف نوع الاستجابة فإذا كانت الاستجابة في شكل نصوص HTML يكون هذا المعامل text/html بينما إذا كان صورة يكون هذا المعامل image/jpeg .

وبالنسبة لصفحات JSP يستطيع المستعرض أن يخمن نوع الاستجابة عن طريق امتداد الملف ، فإذا كان امتداد الملف .htm أو .html يكون ملف نصي HTML بينما إذا كانت صورة في ملف امتداده .jpg تدل على وجود صورة JPEG لكن بالنسبة للسيرفلت فإن المستعرض لا يستطيع معرفة محتوياتها لذلك يعتمد على أن يخبره السيرفلت بنوع بيانات الاستجابة .

شاهدت في الأمثلة السابقة كيف استخدمت العبارة setContentType لتحديد نوع بيانات الاستجابة في الكائن ServletResponse كما يمكنك إذا كنت تعرف طول البيانات المرسلة أن تقوم بتحديد الطول عن طريق العبارة setContentLength ولكن إذا كانت بيانات الاستجابة نصيه فيمكن للمستعرض تقدير طول هذه البيانات بمفرده .

تأكد أيضا من تحديد المعامل content type قبل أن تقوم بالنداء على الوسيلة getWriter لأن برنامج مزود الويب يعتمد على هذا المعامل في تحديد نوع الحروف المستخدمة ، إذا كنت ترسل استجابة نصية مثل نصوص html أو xml فيجب استخدام الكائن PrinterWriter الذي يتم إرجاعه عن طريق الوسيلة response.getWriter ، أما إذا كنت ترسل

بيانات ثنائية مثل ملفات فيديوأوصور فيجب أن تستخدم الكائن ServletOutputStream الذي يتم إرجاعه من الوسيلة response.getOutputStream .

★ **الكائن Servlet Http :**

يتميز الكائن Servlet Http عن الكائن Servlet Generic في أنه يحتوى على وسائل إضافية تمكننا من التعامل مع جميع حالات الاستجابة (مثل Get وPOST وPUT وغيرها) وخصوصا حالتي الاستجابة GET وPOST التي يتم التعامل معهما عن طريق الوسيلتين doGet وdoPost كما نرى من تعريف الوسيلتين:

```
protected void doGet(HttpServletRequest request,
   HttpServletResponse response)
   throws ServletException, java.io.IOException;
protected void doPost(HttpServletRequest request,
   HttpServletResponse response)
   throws ServletException, java.io.IOException;
```

مقارنة بين السيرفلت وJSP:

فيما سبق استعرضنا لكل من السيرفلت وJSP ولكي تستطيع تحديد متى تستخدم واحده منهم يجب أن تعرف مزايا وعيوب كل طريقه وبذلك يمكنك الاختيار بينهم وفقا لحالة برنامجك.

★ **مزايا وعيوب JSP:**

تعتبر لغة JSP لغة سهله فهي تشبه نصوص HTML مما يمكنك من أن ترسل الكود الخاص بك إلى شخص مثلا لا يعرف غير HTML

وسيكون من السهل عليه تخمين الكود وتعديله بسرعة بعكس لغة الجافا التي لن يكون واضح عادة الكود بها من قبل شخص لا يعرفها.

تم إنشاء لغة JSP لتفصل بين محتويات الصفحة والكود خلف هذه الصفحة مما يسهل عملية تقسيم العمل بين المطور ومصمم الرسوميات في الصفحة مما يجعل عملية التطوير سريعة وإنشاء محتويات ثابتة يمكن استخدامها مرات عديدة كما يخصص لكل شخص عمله .

من المزايا الأخرى لصفحات JSP تظهر عند الحاجة لعمل تعديلات كثيرة في البرنامج فيتم تحميل السيرفلت الخاصة بصفحة JSP في كل مرة يتم فيها التغيير تلقائيا وهذا بعكس السيرفلت الذي يحتاج إلى إعادة تمهيد المزود في كل مرة يتم فيها التغيير في البرنامج حتى ترى هذه التغييرات.

ونظرا لأن صفحات JSP تتحول بعد ذلك إلى سيرفلت فمن الصعب الإشارة إلى عيوب بها غير موجودة في نفس الوقت بالسيرفلت ولكن يمكن الإشارة هنا إلى أنه من السهل حدوث ارتباك في فهم الكود إذا لم تحترس في كتابة كود جافا مع نصوص جافا سكريبت أو نصوص HTML ، فنظرا لإمكانية صفحات JSP أن تحتوى مختلف أنواع الكود والنصوص السابقة قد يكون من الصعب جدا التمييز بين كود الجافا أو النصوص المكتوبة إذا لم تقم بتقسيم مناطق الكود بطريقه واضحة .

*** مزايا وعيوب السيرفلت:**

كقاعدة عامة تعتبر عيوب JSP هي مزايا للسيرفلت وبالتالي لأن السيرفلت هي برمجة لكائنات الجافا فإن الكود يكون مفهوما وواضحا إلا إذا كنت تقوم بطباعة جزء كبير من نصوص HTML أوXML عن طريق العبارة out.print أو out.println .

وبالنسبة لنشر السيرفلت فكما عرفت من المثال السابق يجب أن يتم تجميع برنامج السيرفلت والملفات المساعدة له في حزمة ملف WAR كما يجب تحديد نموذج المسار URL الذي يتم عن طريقه عرض السيرفلت ، بينما بالنسبة إلى JSP فإن كل ما تحتاج إليه هو ملف له الامتداد jsp. مما يجعل عملية النشر أسهل ، ولكن في الحياة العملية عندما يتضمن برنامجك الكثير من الملفات فإن ملف الحزمة يكون له فوائد كثيرة ، وأخيرا بالنسبة لتحميل السيرفلت عند تغييرها فأن معظم مزودات الويب تقوم بذلك تلقائيا ولكن قد تحتاج أحيانا إلى أعادة تمهيد المزود لترى هذه التغييرات.

مما سبق يتضح مزايا JSP عن السيرفلت ولكن في كثير من الأحيان ستقوم باستخدام الاثنين ، فالسيرفلت يستطيع القيام بمهام معالجة البيانات من الاتصال بقاعدة البيانات وإجراء حسابات معينة على البيانات ، أما JSP فموجه لإظهار ناتج هذه العمليات بطريقة أكثر تحكما مما يجعل الاستغناء عنه أمرا صعبا. ولكن كقاعدة عامة يفضل استخدام السيرفلت فقط إذا كنت تنوى طباعة بيانات ثنائية (صور أو ملفات فيديو) واستخدم JSP إذا كنت تنوى طباعة نصوص وتعديلها ، ولكن إذا كنت تريد القيام بإظهار نصوص بطريقه تلقائية بدون تدخل من المستخدم وطباعتها فاستخدم السيرفلت.

ويمكنك استخدام صفحات JSP بدون وجود كود لغة جافا في داخلها فهي تحتوى على الكثير من الوسوم التي تمكنك من الوصول إلى كائنات الجافا مباشرة وأيضا على لغة خاصة تسمى لغة التعبير (Expression Language).

وإذا كنت تحتاج المزيد من التحكم في البرنامج يجب أن تختار الطريقة التي تلاءم متطلباتك ونوع البيانات المستخدمة.

الفصل الثالث

التعامل مع النماذج FORMS

النماذج Forms هي الوسيلة التي عن طريقها يمكن الحصول على بيانات من المستخدم ، وسنقوم الآن بعمل نموذج بسيط بحيث يمكن تعديله فيما بعد مع نماذج عملية أكثر.

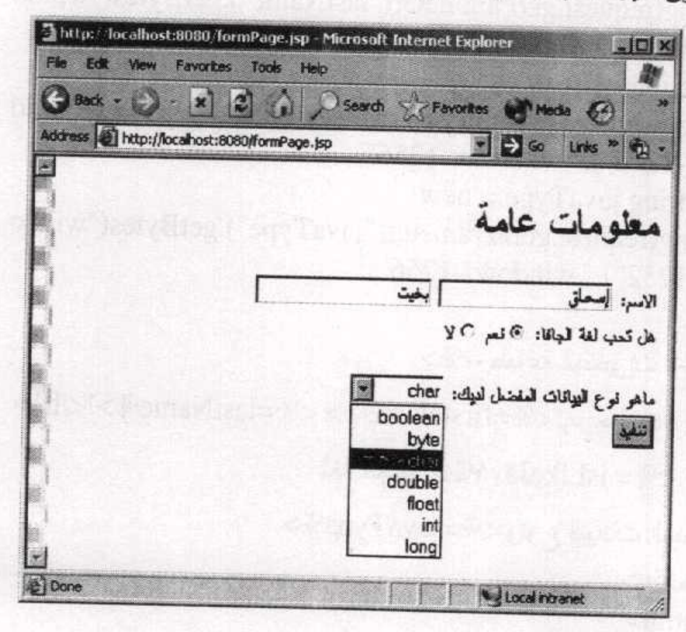
مثال:

هذا المثال يعطى للمستخدم إمكانية إدخال بيانات شخصية في الصفحة مثل الاسم والنوع ، قم بعد كتابة النص بحفظه في الدليل ROOT بالاسم Page.jsp form

```
<%@ page contentType="text/html; charset=windows-
1256" %>
<html dir="rtl">
<body>
<h1>معلومات عامة</h1>
<form action="responsePage.jsp" method="get">
الاسم: <input type="text" name="firstName">
   <input type="text" name="lastName"><br>
هل تحب لغة الجافا:
   <input type="radio" checked name="isLikeJava"
value="تحب">نعم
   <input type="radio" name=" isLikeJava " value=" لا
تحب">لا
<p>
ماهونوع البيانات المفضل لديك:
<select name="javaType">
   <option value="boolean">boolean</option>
   <option value="byte">byte</option>
   <option value="char" selected>char</option>
```

```
    <option value="double">double</option>
    <option value="float">float</option>
    <option value="int">int</option>
    <option value="long">long</option>
</select>
<br>
<input type="submit" value="تنفيذ">
</form>
</body>
</html>
```

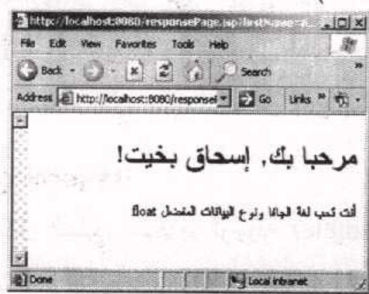
لاحظ أننا قمنا بإنشاء النموذج عن طريق نص HTML فإذا وجدت هذا النص غريبا عليك فيجب أولا مراجعة كتاب في نصوص HTML ، وتكون نتيجة تنفيذ النص السابق كما بالشكل:



قم الآن بإنشاء ملف جديد أسمه "responsePage.jsp" وأكتب الكود الآتي به:

```
<%@ page contentType="text/html; charset=windows-
1256" language="java" import="java.text.*,java.util.*"
%>
<html dir="rtl">
<body>
<%
// الوصول إلى المتغيرات
  String firstName = new
String(request.getParameter("firstName").getBytes("wind
ows-1252"), "windows-1256");
  String lastName = new
String(request.getParameter("lastName").getBytes("windo
ws-1252"), "windows-1256");
  String isLikeJava = new
String(request.getParameter("isLikeJava").getBytes("wind
ows-1252"), "windows-1256");
  String javaType = new
String(request.getParameter("javaType").getBytes("windo
ws-1252"), "windows-1256");
%>
<%-- طباعة المتغيرات --%>
<h1>مرحبا بك, <%=firstName%> <%=lastName%>!</h1>
أنت <%= isLikeJava%> لغة الجافا
ونوع البيانات المفضل <%=javaType%>
</body>
</html>
```

وإذا نفذت كل خطوة بطريقة صحيحة فيجب أن ترى الشكل الآتي بعد تسجيل اسمك واختيار النوع ونوع البيانات المفضل (إذا كانت هناك مشاكل فتابع الملاحظات القادمة):



- لاحظ أن لغة الجافا لغة حساسة لحالة الأحرف فمثلا النوع String الذي هو من أنواع المتغيرات التي تقبل قيم نصية (حروف وأرقام) يجب أن تكتب أول حرف كبير والباقي صغير وإلا أعطى المترجم خطأ أثناء التنفيذ ، بالمثل اسم المتغير نفسه المستخدم في النموذج يجب أن يكون مطابق فمثلا firstName لا يساوى المتغير firstname . اعلم أن هذا يبدو روتيني جدا ولكن هذا يمنح اللغة قوة أكثر ستكتشفها فيما بعد.
- لاحظ أنه بالنسبة للغة العربية فإن المتغير String يستخدم افتراضيا نظام تكويد قياسي يسمى UTF-8 وهذا لا يناسب حروف العربية لذلك يجب أنشاء متغير نص جديد واستخدام نظام التكويد "windows-1256".
- في المثال السابق الخاص بالصفحة "responsePage" ستجد أن كل العبارات المستخدمة مألوفة فلديك مثلا العبارة request وهي

من الكائنات المتضمنة في لغة JSP وتستخدم للحصول على قيم متغيرات النماذج في الأساس.

- الكائن request هو في الواقع نسخه من كائن الجافا HttpServletRequest والذي يمكن استخدامه بنفس الطريقة في السيرفلت وسنتعلم الآن استخدام الكائن request.

★ استخدام الكائن request:

لاحظنا في المثال السابق استخدام الوسيلة getParameter مع الكائن request والذي يستخدم لإرجاع قيمة متغير من نموذج ، وكما ذكرنا يجب أن يكون المتغير بنفس حالة الأحرف ، وإذا طلبت مثلا متغير غير موجود فإن الوسيلة ترجع القيمة null وهذه القيمة تعبر عن نص فارغ لأن الوسيلة getParamete دائما ترجع قيمة نصية فإذا كنت تريد التعامل مع القيمة كعدد عشري يجب عليك تحويلها بنفسك باستخدام أحد دوال التحويل.

لاحظ أن القيمة null تختلف عن القيمة "" والتي تعبر عن وجود المتغير ولكن لا يوجد به نص حيث عدد الحروف به = 0 أما null فتعبر دائما عن عدم وجود المتغير.

من المثال السابق في الفقرة السابقة استخدام الطريقة GET لتمرير متغيرات النموذج ، وفي الواقع يوجد طريقه أخرى وهي الطريقة POST والفرق بينهم هو أن الطريقة GET تقوم بتمرير المتغيرات في سطر المنوان URL نفسه للصفحة ، أما الطريقة POST فتقوم بتمرير المتغيرات في الصفحة نفسها كجزء من الكائن request .

بالتالي يمكنك ملاحظة سطر العنوان URL في الصفحة "responsePage.jsp" والذي سوف يحتوى على علامة الاستفسار ?

متبوعة باسم المتغير وقيمته هكذا first Name=JSP متبوعة بعلامة & للربط بين المتغيرات في سطر واحد فيمكنك حتى كتابة هذا السطر مباشرة وتنفيذ الصفحة بدون الرجوع لصفحة النموذج الأولى.

http://localhost:8080/SimpleFormHandler.jsp?firstName=Isaac
&lastName=Bekheet&sex=male&javaType=float

لاحظ بالنسبة للغة العربية ستجد أن قيم المتغير تكون مشفرة هكذا حتى يتم تحويل الحروف بصورة صحيحة %C7%D3%CD%C7%DE ويتم تمثيل ذلك عن طريق العلامة % متبوعة بكود الحرف بالنسبة للنظام السداسي عشر.

وميزه هذه الطريقة أنك تستطيع في وقت تريد اختبار عمل الصفحة أن تقوم بتغيير جزء من هذه المعاملات ولن تحتاج إلى إدخال البيانات كلها في صفحة النموذج في كل مرة تحتاج إليها إلى اختبار الصفحة ، أيضا نظرا لأن المعاملات تكون في سطر العنوان URL فإن نتيجة الصفحة يمكن حفظها وتخزينها ضمن مجموعة Favorites ببرنامج Internet Explorer مما يتيح لك الرجوع دائما إلى الصفحة في أي وقت ، ولكن هذا ليس الحال بالنسبة لاستخدام الطريقة POST

ويظهر عيب الطريقة GET عند تمرير عدد كبير من المعاملات والحروف فيتم قطعها عند الحد 4KB حرف مما يمنع مثلا تمرير نص كبير ، لذلك لابد من استخدام الطريقة POST هنا.

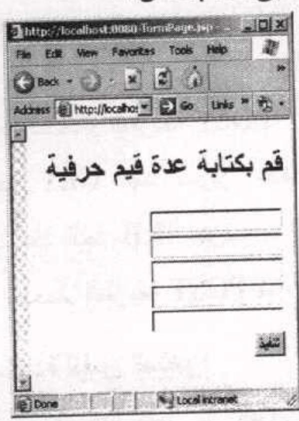
★ التعامل مع قيم متعددة لنفس المتغير:

عندما يتم تسمية عدة حقول من حقول النموذج بنفس الاسم فإن المستعرض يقوم بتمرير عدة قيم لنفس المتغير ، ويمكننا التعامل مع هذا

الوضع كما بالمثال التالي:

```
<%@ page contentType="text/html; charset=windows-
1256" %>
<html dir="rtl">
<body>
<h1>قم بكتابة عدة قيم حرفية</h1>
<form action="responsePage.jsp" method="get">
   <input type="text" name="names"><br>
   <input type="text" name="names"><br>
   <input type="text" name="names"><br>
   <input type="text" name="names"><br>
   <input type="text" name="names"><br>
   <input type="submit" value="تنفيذ">
 </form>
 </body>
 </html>
```

وتكون نتيجة النص السابق كما بالشكل:

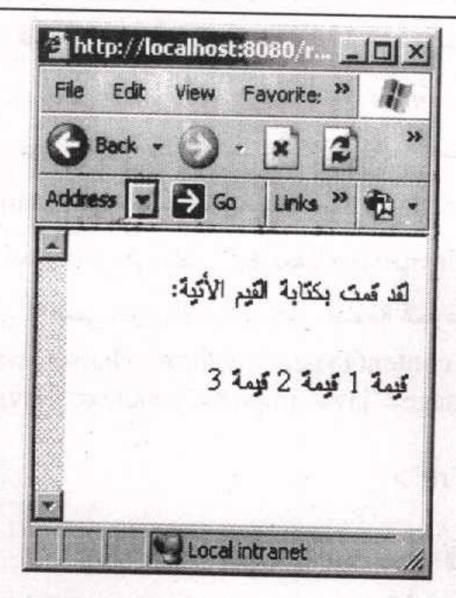


لاحظ من كود النص السابق أننا كتبنا نفس الاسم لحقل النصوص text field لذلك لن تعمل الوسيلة getParameter بطريقة صحيحة لأنها سوف ترجع قيمة أول حقل فقط ولكن يجب استخدام الوسيلة getParameterValues كما سوف يتضح:

قم الآن بكتابة النص الآتي في ملف "responsePage.jsp" (يمكنك تغيير اسم الملف ولكن لا تنسى تغيير الاسم أيضا في صفحة النموذج) :

```
<%@ page contentType="text/html; charset=windows-
1256" language="java" import="java.text.*,java.util.*"
%>
<html dir="rtl">
<body>
<p>لقد قمت بكتابة القيم الآتية:</p>
<p> <%
// الوصول إلى قيم المصفوفة من متغير النموذج
  String names[] = request.getParameterValues("names");
  for (int i=0; i < names.length; i++)
  {
     out.println(new String(names[i].getBytes("windows-
1252"), "windows-1256"));
  }
%>
</p>
</body>
</html>
```

قم الآن بفتح الصفحة الأولى "formPage.jsp" واكتب عدة قيم في الحقول المختلفة ولا يشترط ملء الحقول كلها ثم أضغط على مفتاح "التنفيذ" ويجب أن تكون النتيجة مشابهة للشكل:



لاحظ هنا أننا استخدمنا الوسيلة getParameterValues وهي تقوم بإرجاع مصفوفة حجمها يتحدد حسب عدد المتغيرات القادمة من النموذج وحتى إذا كان هناك متغير واحد فقط فأنها تقوم بإرجاع مصفوفة أحادية البعد .

- عادة تكون أسماء متغيرات النموذج معلومة ولكن في حالة عدم معرفتها يمكنك الوصول إلى هذه الأسماء عن طريق استخدام الوسيلة getParameterNames وتعريف هذه الوسيلة في الكائن request يكون على الشكل:

java.util.Enumeration getParameterNames()

حيث يعبر النوع Enumeration عن عدة كائنات نصية تعبر عن أسماء متغيرات النماذج ، وإذا كنت خمنت يمكنك استخدام هذه الوسيلة مع

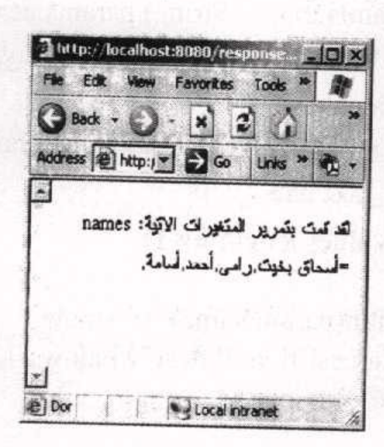
الوسيلة getParameter أو getParameterValues لإظهار أسماء وقيم المتغيرات.

وفيما يلي مثال على ذلك قم باستبدال كود الصفحة "responsePage.jsp":

```
<%@ page contentType="text/html; charset=windows-
1256" language="java" import="java.text.*,java.util.*"
%>
<html dir="rtl">
<body>
لقد قمت بتمرير المتغيرات الآتية:
<%
// الوصول إلى أسماء المتغيرات وتخزينها في المتغير params
   java.util.Enumeration params =
request.getParameterNames();
   while (params.hasMoreElements())
   {
// تخزين كل الأسماء عن طريق حلقة تكرار
      String paramName = (String) params.nextElement();
// إذا كان هناك أكثر من معامل أستخدم الوسيلة getParameterValues
      String paramValues[] =
         request.getParameterValues(paramName);
// إذا كان هناك معامل واحد فقم بطبعه
      if (paramValues.length == 1)
      {
         out.println(paramName+"="+(new
String(paramValues[0].getBytes("windows-1252"),
"windows-1256")));
      }
```

```
        else
        {
// لعدة متغيرات فقم باستخدام حلقة التكرار
            out.print(paramName+"=");
            for (int i=0; i < paramValues.length; i++)
            {
// إذا لم تكن هذه أول حلقة تكرار فقم بطباعة فاصلة بين القيم وبعضها
                if (i > 0) out.print(',');
                out.print(new
String(paramValues[i].getBytes("windows-1252"),
"windows-1256"));
            }
           out.println();
        }
    }
%>
</body>
</html>
```

ويجب أن تكون النتيجة قريبة للشكل التالي:



يمكنك تجربة هذا المثال بدون استخدام صفحة النموذج ولكن بتمرير المعاملات مباشرة في سطر العنوان URL مباشرة كما ذكرنا .

★ النماذج من خلال السيرفلت:

كما عرفت من الأمثلة السابقة أنه يمكننا الحصول على البيانات من النماذج عن طريق الوسيلة Get do إذا كان النموذج يستخدم الطريقة GET لإرسال البيانات عبر برتوكول HTTP ولكن بالنسبة للطريقة POST فيجب أن نقوم بتعريف الوسيلة depots التي يتم استدعائها من المستعرض في هذه الطريقة . أيضا يمكننا تعريف الوسيلة service التي يمكنها استقبال بيانات من النموذج سواء استخدم النموذج الطريقة GET أو الطريقة POST وهذا لا يجعل السيرفلت محدودة لطريقة عمل النموذج . وتقوم أيضا الوسيلة service باستخدام معاملين هما HttpServletRequest و HttpServletResponse مثل الوسيلتين do Get و Post do ، ولكن لاحظ هنا أن المعامل Http Servlet Request مثله مثل الكائن request المستخدم مع JSP ، وهذا يعنى أنك تستطيع أن تقوم بالوصول إلى بيانات النموذج بنفس الطريقة المستخدمة مع JSP

يمكنك أيضا أن تقوم بتعديل الوسيلة Post do لتعالج كلا الطريقتين POST و GET كما يلي :

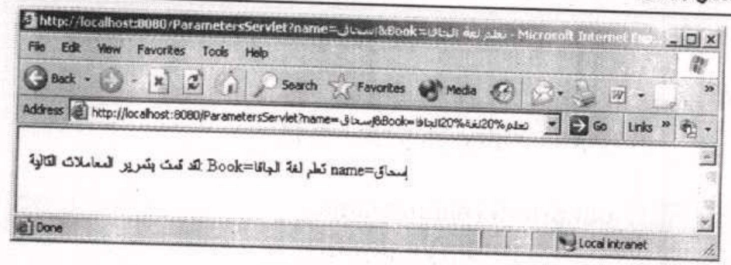
```
public void doPost(HttpServletRequest req,
HttpServletResponse res)
                    throws ServletException, IOException
{
doGet(req, res);
}
```

وفيما يلي مثال على استجابة السيرفلت للطلب من النموذج وإظهار قيم المعاملات في الصفحة كما حدث في المثال السابق الذي استخدمنا فيه كود SP :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class ParametersServlet extends HttpServlet
{
  public void service(HttpServletRequest request,
    HttpServletResponse response)
    throws IOException
  {
//HTML تحديد للمستعرض نوع البيانات على انه
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<body>");
    out.println("لقد قمت بتمرير المعاملات التالية:");
   // تحديد أسماء المعاملات
    Enumeration params = request.getParameterNames();
    while (params.hasMoreElements())
    {
// الحصول على أسم ثاني معامل
       String paramName = (String)
params.nextElement();
//استخدام الوسيلة getParameterValues في حالة وجود أكثر من معامل
       String paramValues[] =
```

```
            request.getParameterValues(paramName);
// إذا كان هناك قيمة واحدة يتم طباعتها
        if (paramValues.length == 1)
        {
            out.println(paramName+
                "="+paramValues[0]);
        }
        else
        {
// إذا كان هناك أكثر من قيمة يتم استخدام حلقة التكرار
            out.print(paramName+"=");
            for (int i=0; i < paramValues.length; i++)
            {
// طباعة فاصله بين القيم عدا أول قيمة
                if (i > 0) out.print(',');
                out.print(paramValues[i]);
            }
            out.println();
        }
      }
    out.println("</body>");
    out.println("</html>");
  }
}
```

قم الآن بترجمة الملف عن طريق المعالج javac ثم قم بإنشاء ملف web.xml واختبر السيرفلت عن طريق وضع معاملات في شريط العنوان كما يتضح من الشكل التالي:



★ الطرق المختلفة للنماذج:

هناك أربع طرق يمكنك استخدامهم لإنشاء النماذج يمكن تلخيصهم كما يلي:

1. نموذج ثابت عن طريق نصوص HTML .
2. نموذج منفصل عن طريق لغة JSP.
3. نموذج واحد للتعامل مع الإدخال ومعالجة المتغيرات عن طريق لغة JSP.
4. نموذج يستخدم به JSP والسيرفلت معا.

لقد قمنا في أول الفصل بعرض مثال لنموذج ثابت عن طريق نصوص HTML ومعالجة متغيرات النموذج عن طريق صفحة بها لغة JSP وتسمى ثابتة لأنك تقوم بتصميم الصفحة واختيار نوع الكائنات التي يتم وضعها في الصفحة مثل الحقول النصية ، ولكن يمكن في الحالة الثانية عند استخدام لغة JSP أنشاء نموذج ديناميكي بحيث يتم مثلا عند قراءة ملف معين أو قاعدة بيانات أن يتم تحديد الكائنات التي ستوضع في الصفحة فتذكر دائما أنك عند استخدامك للغة JSP أنك تمتلك إمكانيات لغة الجافا كلها.

ولكن لأن الطريقة الثالثة وهي استخدام نفس الصفحة للنموذج وللتعامل مع متغيرات النموذج بها بعض الأفكار الجديدة سنقوم الآن بشرحها:

- **استخدام نفس الصفحة للتعامل مع مدخلات ومخرجات النماذج**

تستخدم هذه الطريقة عندما يحتاج الزائر إلى تصفح نفس الصفحة عدة مرات مثلا عند رؤية نتائج البحث بعد كتابة القيمة المراد البحث عنها ، ويمكنك تحديد النموذج بعد تنفيذه بعدة طرق هي:

- اختبار وجود متغيرات النموذج أم لا.
- تمرير متغير خاص للدلالة على تنفيذ النموذج من قبل الزائر.
- تنفيذ النموذج عن طريق الوسيلة POST واختبار الكائن request في الصفحة.

ويمكنك تمرير قيمة مبدئية للصفحة في حقل مخفي بحيث يمكنك فيما بعد اختبار قيمة هذا الحقل ومعرفة ما إذا تم مثلا تنفيذ النموذج أم لا ، وصيغة هذا الحقل يمكن أن تكون على الصورة الآتية:

```
<"input type="hidden" name="isSubmitted" value="yes>
```

وفي أول مرة يزور فيها المستخدم الصفحة لن ترى هذا المتغير لأن المستخدم لم يقوم بتنفيذ النموذج بعد ويمكنك اختبار وجود المتغير أم لا عن طريق السطر

```
request.getParameter("isSubmitted");
```

أيضا يمكنك مقارنة القيمة التي تم إرجاعها فإذا كانت null فإن النموذج لم يتم تنفيذه بعد .

وكما ذكرنا أن الطريقة الأخرى هي استخدام الوسيلة POST ويمكن اختبار تنفيذ النموذج عن طريق الوسيلة request.getMethod() حيث يمكن استخدامها كما يلي:

قم بالإعلان عن النموذج باستخدام الطريقة POST

```
<form action="anyform.jsp" method="POST">
```

ثم قم باستخدام الوسيلة request.getMethod لمعرفة ما إذا تم تنفيذ النموذج أم لا.

```
if (request.getMethod().equals("POST")) {
 //أكتب كود التعامل مع المتغيرات هنا
}
```

الفصل الرابع

برتوكول النقل HTTP

قمت في الفصول السابقة باستخدام تقنية تسمى الطلب (من المتصفح أو الزبون) والاستجابة (من المزود أو السيرفر) وهى من العمليات الأساسية للغة JSP والسيرفلت . وتم استخدام الكائن out ضمنيا في هذه العملية بدون أن تدرى كما تم إنشاء الكائن HttpServletResponse واستخدام وسيلته getWriter في عملية الاستجابة ، كذلك قمت بالوصول إلى متغيرات النماذج عن طريق الكائن request .

وتستخدم عادة السيرفلت مع برتوكول النقل HTTP وهو اختصار للكلمة (Hypertext Transfer Protocol) لذلك يوجد الكائن HttpServlet وسنقوم الآن بشرح لهذا البرتوكول.

داخل برتوكول النقل HTTP:

يتصل المتصفح في جهاز الزائر بجهاز المزود غالبا عن طريق برتوكول HTTP ولكن في بعض الأحيان يتصل به عن طريق برتوكول نقل الملفات FTP ، ويمكنك كمبرمج جافا استخدام الفئة URL والفئة URLConnection لمعالجة الاتصال من ناحية جهاز الزائر والسيرفلت وJSP من ناحية جهاز المزود.

ملحوظة: يقصد بكلمة فئة (*CLASS*) هي مجموعة من الخواص *PROPERTIES* والوسائل *METHODS* والأحداث *EVENTS* ومتغيرات تنتمي لهذه الفئة *MEMBER VARIABLES* وهى من الوحدات الأساسية في البرمجة المتجهة *OOP* وعندما تقوم بإنشاء نسخة من الفئة تسمى هذه النسخة كائن *OBJECT* ويتم توريث كل سمات الفئة لهذا الكائن ولكن الفرق بين الفئة والكائن فرق نظري ويمكننا اعتبار كل فئة هي كائن وسترى خلال هذا الكتاب الكثير من الأمثلة التي ستعطيك فكرة واضحة عن الكائنات عمليا.

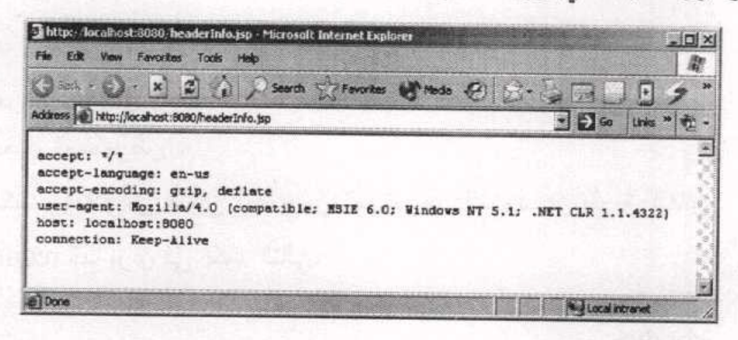
ويعتبر بروتوكول HTTP في الواقع منفذ من منافذ جهاز الكمبيوتر وعادة يستمع جهاز الكمبيوتر للطلبات requests من الإنترنت عن طريق منفذ 80 وبعد أن يتم الاتصال يتم إرسال بعض السطور من النصوص التي عن طريقها يعرف المزود أي صفحة يريد جهاز الزائر أن يراها كما يتم إرسال معلومات أخرى مفيدة عن جهاز الزائر مثل أصدارة المتصفح ونوعها (Internet Explorer, Mozilla, Opera...) واللغة التي يستخدمها والبلد وأيضا مزود خدمة الإنترنت المتصل به . ويعتبر أول سطر في نص الطلب (request) المرسل إلى المزود هو السطر الهام حتى يعرف المزود الملف المطلوب أما باقي النص فيمكن فهمه بسهولة حتى أنك تستطيع استخدام أداة مثل telnet للتعامل يدويا مع المزود بدون استخدام المستعرض .

ويمكننا رؤية رأس عنوان الطلب من المستعرض للمزود عن طريق الكائن request كما نرى في الكود التالي:

```
<html>
<body>
<pre>
<%
   java.util.Enumeration e = request.getHeaderNames();
   while (e.hasMoreElements())
   {
      String headerName = (String) e.nextElement();
      out.print(headerName+": ");
      java.util.Enumeration h =
request.getHeaders(headerName);
      while (h.hasMoreElements())
      {
```

```
        String header = (String) h.nextElement();
        out.print(header);
        if (h.hasMoreElements()) out.print(", ");
      }
      out.println();
    }
%>
</pre>
</body>
</html>
```

وتكون النتيجة هي رؤية المخرجات التالية:



لاحظ من الشكل السابق عدة عناصر ثابتة يحتوى عليها رأس العنوان وفيما يلي شرح لهذه العناصر :

★ **العنصر Accept:**

تشير العبارة Accept هنا على نوع المحتويات التي يستطيع المستعرض أن يقبلها وأحيانا تظهر قائمة بالملفات التي يفضلها المستعرض text/xml أو text/html ، وعادة ستجد العلامة */* في أخر القائمة مما يدل على أن المستعرض يقبل كل شيء ولكنه يفضل الملفات التي بالقائمة

وتختلف هذه القائمة باختلاف المستعرض (Internet Explorer, Mozilla, Netscape…)

كمثال أخر لرأس العنوان هو ما ترسله التليفونات المحمولة فنجد أنها تفضل الملفات من نوع html أو الصور bitmaps كما يتضح من النص التالي:

```
Accept: application/x-hdmlc, application/x-up-alert,
application/x-up-cacheop,
  application/x-up-device, application/x-up-digestentry,
text/x-hdml;
      version=3.1,
  text/x-hdml;version=3.0, text/x-hdml;version=2.0,
text/x-wap.wml,
  text/vnd.wap.wml, */*, image/bmp, text/html
```

✶ العنصر Accept-Language:

يعرض هذا العنصر اللغة الخاصة بالمستخدم ، فمثلا إذا كانت الانجليزية قد ترى الحرف en أو en-us للولايات المتحدة ويمكن استخدام هذا العنصر مثلا لمعرفه اللغة التي يفضلها المستخدم وعرض الصفحة التي تناسبه.

✶ العنصر Accept-Charset:

إذا وجدت هذا العنصر فإنه يشير إلى نوع قاعدة الحروف المستخدمة فمثلا بالنسبة للمستعرض نتسكيب Netscape قد ترى العنصر بالشكل التالي:

```
Accept-Charset: iso-8859-1,*,utf-8
```

★ العنصر User-Agent:

يعتبر من أهم عناصر رأس العنوان حيث يمكننا معرفة نوع المستعرض الذي يقوم بالطلب ولكن الشيء الغريب أنك ستجد المستعرض Internet Explorer والمستعرض Netscape يعرف كل منهم نفسه على أنه مستعرض من نوع يسمى Mozilla والذي تم إنشاء الكود الخاص به من أجل المستعرض Netscape 5.0 ويأتي أسم Mozilla من المطورين الأصليين للمستعرض القديم Mosaic ثم تم تطوير هذا المستعرض لإنشاء المستعرض Netscape Navigator الذي اعتبروه وحش برامج استعراض النت Godzilla Mosaic أوMozilla ، هذا سبب أيضا أن المستعرض Internet Explorer قبل انتشاره كان عليه أن يحدد أنه متوافق مع المستعرض Mozilla ورغم انتشاره الآن إلا أنه مازال يعرض نفس القيمة ولكن إذا أردت تحديد ما إذا كان المستعرض نوعه Netscape أو Internet Explorer فستجد أن الأخير يقوم بإظهار القيمة MSIE وبالتالي يمكن كتابة كود التالي لتحديد النوع :

```
if (request.getHeader("USER-AGENT").
  indexOf("MSIE") >= 0)
{
  //Internet Explorer  كود خاص بالمستعرض
}
else
{
  //Netscape  كود خاص بالمستعرض
}
```

✱ عناصر الاستجابة:

أما فيما يتعلق بحالة الاستجابة response فيمكننا التركيز في بعض العناصر الهامة التي يقوم المزود بإرسالها ويمكن الاستفادة منها عند كتابة كود جافا .

✱ العنصر **Content-Type**:

من أهم العناصر التي يعرف عن طريقها المستعرض نوع البيانات التي سيقوم بعرضها – وعادة تكون text/html – ولكن يمكنك كما تعلمت سابقا أن تقوم بتغيير ذلك مثلا إذا استخدمت بيانات xml يجب أن تكون text/xml.

✱ العنصر **Content-Length**:

من العناصر الحيوية للمستعرض خصوصا إذا كانت استجابة المزود تحتوى على بيانات ثنائية حيث يدل هذا العنصر على عدد البايت في الاستجابة مما يعطى التأكد للمستعرض من قراءة الرسالة كاملة.

✱ العنصر **Cache-Control**:

يحدد هذا العنصر المقدار الذي تبقى فيه الصفحة مخبأه وفى هذه الحالة يتم تحديد المدة بالثواني للصفحات قبل إزالتها وبالتالي تكون هناك فائدة لإعادة تحميل نفس الصفحة بسرعة عند الطلب وافتراضيا لا تكون صفحات JSP مخبأه ولكن يمكن تحديد ذلك عن طريق العبارة.

Cache-Control: maxage=180

يتم عن طريق العبارة السابقة تحميل نسخة مخبأة من الصفحة لمدة 3 دقائق (180 ثانية) ولا تؤدى العبارة السابقة إلى إعادة تحميل الصفحة من

تلقاء نفسها بعد هذه المدة ولكن يمكنك أداء ذلك عن طريق الوسم Refresh كما يلي:

```
<META HTTP-EQUIV="Refresh" Content = "15;
URL=yourpage.jsp">
```

ويتم تحميل الصفحة yourpage.jsp بعد مضى 15 ثانية

الفرق بين الوسيلة POST والوسيلة GET:

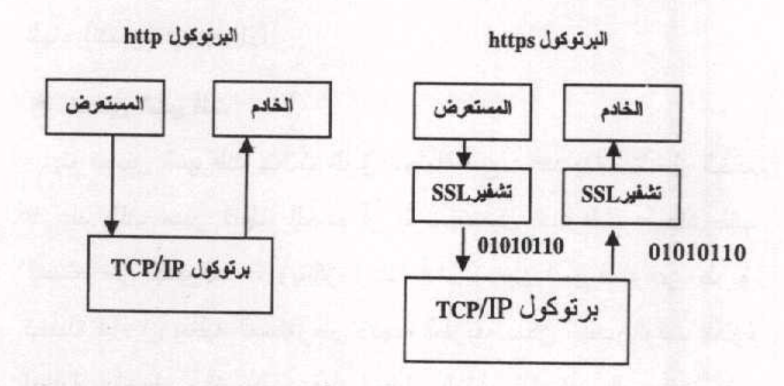
تختلف الوسيلة POST عن الوسيلة GET في أن الأولى يتم إرسال بيانات النموذج بعد أول سطر من سطور الطلب بحيث يتم عرض كل المتغيرات وقيمها بالصيغة var=value ويتم الفصل بين المتغيرات بالعلامة & كما يتم تحديد قيمة العبارة Content-Type لهذه الوسيلة على إنها application/x-www-form-urlencoded .

من هنا يتضح أن استخدام الوسيلة GET أسرع بكثير من إرسال البيانات في رأس الطلب ولكن كما ذكرنا يقابلنا هنا حد طول النص المسموح به في الوسيلة GET وهو 4KB وهذا حتى ليس أكيد لأن بعض المزودات لا تقبل أكثر من 2KB من الحروف وأيضا تظهر ميزة الوسيلة POST في عدم رؤية الزائر للبيانات التي يتم إرسالها إذا كانت بيانات هامة مثل رقم بطاقة الائتمان مثلا وعلى النقيض تكون الوسيلة GET أكثر ميزة عند حفظ نتائج بحث صفحة معينة فيمكن الرجوع إلى نفس النتائج في أي وقت لأن معاملات البحث مخزنة في سطر العنوان URL.

لاحظ هنا أن الوسيلة POST لا تقوم بحماية كاملة للبيانات المرسلة فقد يستطيع الهاكر أو أي شخص محترف أن يعترض هذه البيانات ويحصل عليها لذلك هناك طرق معروفة ومضمونة لتشفير وإرسال

البيانات ذات السرية العالية وقريبا يصدر كتاب لطرق الحماية والتأمين على الإنترنت من دار البراء ليكون متخصص في هذا الموضوع الهام .
ولكن يمكن أن نوضح فكرة التشفير هنا:

قم بالدخول على موقع للشراء وحاول أن تصل حتى خطوة إدخال بيانات كارت الائتمان أنظر إلى سطر العنوان ستجد أن البرتوكول هنا عادة يكون https وليس http والذي عندما يراها المستعرض يفهم أنه يتم استخدام مكتبة التشفير SSL أو(Secure Sockets Layer) وهى تقوم بتشفير البيانات المرسلة والقادمة حتى إذا تم اعتراض هذه البيانات لا يتم معرفة البيانات حيث تظهر على هيئة رموز غير مفهومة وفيما يلي رسم يوضح إرسال واستقبال البيانات عبر http وhttps:



طريقة عمل السيرفلت وJSP

ذكرنا فيما سبق أهم مزايا وخصائص كلا من السيرفلت وJSP وستعرف الآن في هذا الفصل كيف يتم إنشاء وتحميل وإنهاء كلاهما ، فبعكس لغة الجافا العادية التي يكون لنا فيها كامل التحكم في إنشاء كائن أو إزالته من الذاكرة (تدميره) فإن السيرفلت وJSP يتم التحكم فيهما من قبل برنامج المزود الذي يقوم بدون تدخل منك بإنشاء وتحميل أو إعادة تحميل أو تدمير الكائنات عند الطلب بما يحقق أقصى كفاءة لعرض الصفحات للزائر وسنقوم الآن بالشرح التفصيلي لهذه الخطوات:

* طريقة عمل السيرفلت:

يتبع السيرفلت أربع خطوات في طريقة عمله هي على الترتيب:
التحميل Loading – التمهيد Initialization – التنفيذ Execution – إنهاء (تدمير) Cleanup.

* تحميل السيرفلت

يتم تحميل السيرفلت بثلاث طرق محتملة هي : عند بداية تشغيل الخادم – عند طلب مدير النظام للخادم أن يقوم بتحميل السيرفلت – عند طلب المستعرض للسيرفلت ، ويمكن زيادة أداء تحميل السيرفلت عن طريق تحميله قبل أن يطلبه المستعرض وبهذه الطريقة يمكن تجنب الوقت اللازم لعملية التحميل والتمهيد ، فعلى سبيل المثال إذا كان السيرفلت يقوم بالاتصال بعدة قواعد بيانات ويقوم بإحضار بعض البيانات منهم فإن أول مستخدم يقوم بطلب هذه السيرفلت سيعانى من بطء ملحوظ ، لذلك من

الأفضل تحميل السيرفلت مع تشغيل الخادم والذي قد يأخذ وقت أطول ولكن لن يلاحظ الزائر أي تأخير في الاستجابة لطلبه.

ولكي يمكن للخادم أن يصل للسيرفلت ويقوم بتحميله يجب عليه أن يعرف أسم الملف التنفيذي للسيرفلت class.* والذي يكون عادة هو الاسم السيرفلت الذي يطلبه المستعرض وإذا كان السيرفلت مخزن في حزمه فيتم استخدام تسميه التوارث للوصول إليه ، فمثلا إذا كان المستعرض يقوم باستدعاء السيرفلت findBooks الذي يوجد في الحزمة tools فإن أسم السيرفلت بالكامل يكون tools . findBooks.

ويعرف برنامج المزود (Tomcat) أنك طلبت تنفيذ سيرفلت عن طريق العبارة /servlet/ التي يتم تحديدها في سطر العنوان ، فمثلا من المثال السابق يتم تنفيذ السيرفلت عن طريق العنوان :

http://localhost:8080/servlet/tools.findBooks

وبعد استدعاء السيرفلت يقوم برنامج المزود بفحص وجود السيرفلت في الذاكرة فإذا كان غير موجودا فأن المزود يقوم بتحميل السيرفلت عن طريق كائن التحميل ثم يتم التمهيد ويصبح السيرفلت جاهز للاستخدام.

✶ تمهيد السيرفلت:

عندما يتم تمهيد السيرفلت يتم استدعاء الوسيلة init والتي يتم تمرير الكائن ServletConfig بها كمعامل كما يتضح من التعريف التالي للوسيلة init:

```
public void init(ServletConfig config)
  throws ServletException
```

ويحتوى الكائن ServletConfig على معاملات تسمى معاملات التمهيد بالإضافة إلى الكائن ServletContext ويتم تحديد معاملات التمهيد بسهولة عن طريق الملف الوصفي للنشر كما يلي:

```
<servlet>
...
<init-param>
   <param-name>parameter</param-name>
   <param-value>value</param-value>
</init-param>
...
</servlet>
```

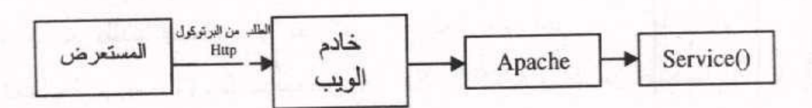
حيث يتم تحديد المعامل مكان العبارة parameter وقيمة هذا المعامل مكان العبارة value ويمكن الوصول إلى هذه المعاملات ومعرفة قيمها عن طريق أحد الوسيلتين التاليين في الكائن ServletConfig

```
public java.util.Enumeration getInitParameterNames()
public String getInitParameter(String name)
```

وهذه المعاملات يمكن استخدامها في كثير من الأشياء مثل تحديد مسار ملفات يحتاج إليها السيرفلت للعمل أما الكائن ServletContext فيستخدم لتبادل المعلومات بين السيرفلت وملفات خارجية

✶ – تنفيذ السيرفلت:

عندما يتم طلب السيرفلت يقوم برنامج المزود باستدعاء السيرفلت والنداء على الوسيلة service وتمرير الكائنين ServletRequest وServletResponse كمعاملات وتتضح أكثر هذه العملية من الرسم التالي:



ملحوظة: منذ الإصدار 1.4 لمكتبة JDK ضمت شركة صن (الشركة التي أنشأت الجافا) مكتبة للتعامل مع البروتوكول المشفر لـ SSL في إصداراتها.

★ إنهاء وتدمير السيرفلت:

يتم إنهاء السيرفلت إذا لم تعد هناك حاجة لاستخدامه مما يخلي الذاكرة من المساحة التي كان السيرفلت يشغلها وقد يتم ذلك عن طريق إغلاق أحد أجهزة الخادم أو برنامج المزود (Tomcat) أو يدويا من مدير النظام فيتم استدعاء الوسيلة destroy التي تقوم بعملية إنهاء السيرفلت من الذاكرة بدلا من انتظار نظام التشغيل لأداء هذه العملية في بطء وعلى سبيل المثال عندما يقوم السيرفلت بفتح أكثر من قاعدة بيانات في نفس الوقت يجب هنا التأكد من النداء على هذه الوسيلة لإغلاق هذه الوصلات وتحرير الذاكرة منها.

طريقة عمل صفحات JSP:

تتضمن طريقة عمل JSP المراحل التالية

المعالجة Compilation – التحميل Loading - التمهيد Initialization – التنفيذ Execution – الإنهاء Cleanup

★ مرحلة المعالجة:

عندما يقوم المستعرض بطلب صفحة JSP يفحص التوم كات هذه الصفحة ليرى ما إذا تم معالجتها أم لا فيقوم بعمل معالجة لها إذا كانت أول

مرة يتم طلب الصفحة أو تم تغيير الصفحة منذ آخر مرة تمت فيها المعالجة ثم يقوم التوم كات بفحص لغة JSP المكتوبة ليتأكد من صحتها ثم يقوم بتحويلها إلى سيرفلت ثم يقوم بمعالجة السيرفلت .

✶ تحميل صفحة JSP:

مثل السيرفلت تماما يتم تحميل صفحة JSP ولكن عندما يتم تغيير الصفحة يقوم التوم كات بإعادة تحميل ملف السيرفلت class.* الخاص بالصفحة مرة أخرى.

✶ تمهيد صفحات JSP:

من غير الموصى به تعديل الوسيلة init أو الوسيلة destroy بالنسبة لـــ JSP ويفضل استخدام الوسيلة ()jspInit عندما تريد تمهيد الصفحة

```
public void jspInit()
```

وبالمثل يمكنك في هذه المرحلة كتابة كود لفتح وصلات لقواعد البيانات أو فتح ملفات أو الوصول إلى بيانات يتم البحث فيها بصورة متكررة .

ويمكن استخدام الكائن config الذي يشير إلى الكائن ServletConfig والكائن application الذي يشير إلى الكائن ServletContext ولكن حتى تستطيع أن تقوم بتحديد معاملات التمهيد مثل السيرفلت يجب تحديد أسم السيرفلت الخاص بالصفحة الذي سيتم توليده عند معالجة صفحة JSP ويمكن استخدام النموذج التالي لتحديد ذلك:

```
<servlet>
  <servlet-name>myPageJSP</servlet-name>
  <jsp-file>/myPage.jsp</jsp-file>
...
</servlet>
```

★ تنفيذ صفحة JSP:

عند طلب الصفحة يتم استدعاء الوسيلة jspSe::vice والتي تأخذ المعاملين HttpServletRequest و HttpServletResponseولأن هذه الوسيلة يقوم المعالج نفسه بتوليدها فلا تقوم أبدا بمحاولة تعديل هذه الوسيلة.

★ إنهاء صفحة JSP:

بالمثل تكون الوسيلة jspDestroy هي المقابل للوسيلة destroy في السيرفلت فعندما تريد كتابة أي إجراء لتحرير الذاكرة – مثلا إغلاق ملفات مفتوحة – قم بكتابة الكود في النموذج التالي:

```
<%!
  public void jspDestroy()
  {
     // الكود الخاص بتحرير الذاكرة
  }
%>
```

★ كيفية أعادة تحميل الملفات الخارجية:

إذا قمت بكتابة كود في الصفحة JSP وهذا الكود يستخدم دالة خارجية مخزنة في ملف من نوع class.* فإن أي تغيير في هذا الملف لن يؤدى إلى إعادة تحميله حتى وان قمت بتغيير في الصفحة JSP التي تستدعيه ، لذلك قد تحتاج إلى إعادة تحميل مزود Tomcat مرة أخرى.

مثال:

سنقوم بالنداء على وسيلة لإظهار رسالة من ملف خارجي وفيما يلي الكود الخاص بصفحة showMsg.jsp:

```
<%@ page contentType="text/html; charset=windows-
1256" language="java" import="myUtil.*" %>
<html dir="rtl">
 <html>
 <body>
   الرسالة هي :  <%= MsgClass.getMessage() %>
 </body>
 </html>
```

وفيما يلي كود ملف الجافا MsgClass.java:

```
package myUtil;
 public class MsgClass
 {
   public static String getMessage()
    {
      return "لم يتم تغيير الملف";
    }
 }
```

وبعد معالجة الملف MsgClass.java تكون نتيجة عرض الصفحة showmsg.jsp كما يلي:

الرسالة هي: لم يتم تغيير الملف

والآن إذا قمت بتغيير الرسالة في الملف MsgClass.java لتكون "بعد تغيير الملف" ثم قمت بإعادة معالجة الملف فإذا قمت بإعادة طلب الصفحة مرة أخرى فلن ترى الرسالة الجديدة لأنه لم يتم أعادة تحميل الملف MsgClass.class قم الآن بإعادة تمهيد المزود وحاول مرة أخرى سترى النتيجة.

الرسالة هي: بعد تغيير الملف

وإذا قمت باستخدام ملفات WAR لنشر ملفاتك فإن برنامج المزود دائما يقوم بإعادة تحميل الملف الموجودة داخل الملف WAR تلقائيا ويجب عليك دائما استخدام أدوات مثل ANT Tool لإنشاء حزم WAR.

*** الكائن ServletContext:**

يستخدم هذا الكائن للوصول إلى موارد البرنامج مثل ملفات خارجية أو صور موجودة بحزمة ملف WAR وعن طريق استخدامه يستطيع المبرمج أن يصل إلى معاملات التمهيد المختلفة .

ويقوم برنامج المزود بإنشاء هذا الكائن عند بداية تحميل البرنامج وعن طريق تعريف الكائن ServletContextListener يمكنك أجراء عمليات متعلقة بتمهيد وإغلاق البرنامج ويوجد بهذا الكائن الوسيلتين التاليين :

```
public void contextInitialized ( ServletContextEvent sce );
public void contextDestroyed ( ServletContextEvent sce );
```

فيقوم المزود باستدعاء الوسيلة contextInitialized عند بداية البرنامج والوسيلة contextDestroyed عند إنهاء البرنامج أما المعامل ServletContextEvent فيقدم طريقة للوصول إلى الكائن servlet context.

مثال:

الكود التالي يقوم بتسجيل بيانات بداية البرنامج وإنهاؤه عن طريق الكائن ServletContextListener واحفظ الكود بملف أسمه MyListner.java.

```
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
public class MyListener
    implements ServletContextListener {
   public void contextInitialized(ServletContextEvent event) {
     ServletContext context = event.getServletContext();
      context.log("Servelt Context بداية إنشاء");
   }
   public void contextDestroyed(ServletContextEvent event) {
      ServletContext context = event.getServletContext();
      context.log("Servlet Context إنهاء");
   }
 }
```

يقوم هذا المثال بالوصول إلى الكائن ServletContext ثم يقوم بكتابة معلومات إلى ملف التتبع log file الخاص ببرنامج المزود وفيما يلي كود يؤدى نفس الوظيفة بلغة JSP أحفظه بالاسم traceApp.jsp:

```
<%@ page contentType="text/html; charset=windows-1256"%>
<html dir="rtl">
<body>
 <%!
    public void jspInit()
    {
       log("jspInit تم استدعاء الوسيلة ");
    }
 %>
 <%!
    public void jspDestroy()
```

```
    {
       log("jspDestroy تم استدعاء الوسيلة ");
    }
  %>
    log(" jspService تم استدعاء الوسيلة");
  %>
  The current date is: <%= new java.util.Date() %>
  </body>
  </html>
```

لاحظ أن الوسيلة log غير موجودة بلغة JSP ولكن لأن صفحة JSP تترجم بعد ذلك وتحول إلى سيرفلت فهذه وسيلة للكائن GenericServlet الذي يعتبر الكائن الأبوي (superclass) للسيرفلت الخاص بصفحة JSP ، ويجب أن يعرف المزود اسم برنامج الجافا عند نشره فيجب كتابة ملف web.xml على الشكل التالي:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <display-name>TraceApp</display-name>
    <description>view application events </description>
    <listener>
      <listener-class>
        MyListener
      </listener-class>
    </listener>
 </web-app>
```

لاحظ أننا لو استخدمنا حزمة في برنامج الجافا وليكن أسمها MyUtil فيجب أن نكتب أسم الملف هكذا MyUtil.MyListner

لاحظ أيضا أننا قمنا لأول مرة باستخدام ملف web.xml مع صفحة JSP وهذا لأن المزود يحتاج إلى هذا الملف لتحميل ServletContextListener وستجد انه يختلف فقط عن الملف المعتاد في الوسم <listner> </listner> الذي يعرف فيه اسم الكائن الذي يستجيب لإحداث برنامج الويب الخاص بنا .

قم الآن بحزم البرنامج بعد ترجمته ولضرورة أن تكون الأدلة على الشكل التالي قم بإنشاء ملف WAR:

traceAPp.jsp
WEB-INF/web.xml
WEB-INF/classes/MyListener.class

ثم أكتب الأمر

jar cfv traceApp.war *

قم الآن بنشر الملف عن طريق Tomcat manager الذي يتيح لك أيضا إيقاف البرنامج وتستطيع مشاهدة بيانات إنهاء البرنامج ويجب أن تستخدم العنوان URL التالي للوصول إلى الصفحة

http://localhost:8080/traceApp/traceApp.jsp

لاحظ هنا أننا وضعنا المسار /traceApp/ لأن المزود يحتاج إلى هذا المسار الخاص باسم ملف WAR والذي يسمى Context Path

ستجد أن الملف Log الذي به البيانات الخاصة بالبرنامج في الدليل \logs الخاص بـــ Tomcat ويكون الملف بالاسم locaihost_log.date.txt ، حيث أن العبارة date تدل على التاريخ الذي تم فيه أنشاء الملف ، قم الآن بفتح الملف وتتبع تسلسل الأحداث منذ إنشاء البرنامج وحتى إنهاؤه.

الفصل الخامس

الكائنات الأساسية للسيرفليت

سنقوم معا في هذا الفصل بعرض تفصيلي للكائنات الأساسية الموجودة بأي سيرفلت وهي:

- الكائن JspWriter
- الكائن PageContext
- الكائن JSPEngineInfo

الكائنات المتضمنة في JSP:

تقدم صفحات JSP بنية بسيطة فوق البنية الخاصة بدوال السيرفلت API والغرض منها جعل أنتاج كود HTML للصفحة بسيط وسهل بحيث يمكن الوصول إلى المعلومات الخاصة بحالة الطلب request أو الاستجابة response أو المتغيرات المحفوظة session.

وفيما يلي جدول لهذه الكائنات والفئة المعرفة بها (الكائنات الأبوية).

كائن JSP	الفئة
request	HttpServletRequest
response	HttpServletResponse
out	JspWriter
session	HttpSession
application	ServletContext
pageContext	PageContext
config	ServletConfig
page	Object
exception	Throwable

يتم إنشاء الكائنات السابقة عند تنفيذ صفحة JSP ويمكن استخدامهم للوصول إلى كائنات السيرفلت ومن الفصول السابقة يجب أن تكون قد تعرفت على معظم هذه الكائنات وفيما يلي ملخص لكل كائن منهم:

✶ الكائن **request:**

هذا الكائن مسئول عن حالة الطلب وينحدر من الكائن ServletRequest وفى معظم الحالات يكون هذا الكائن هو نسخة فرعية من الكائن HttpServletRequest إذا كنت تستخدم برتوكول HTTP أما إذا كنت تستخدم برتوكول آخر فستجد الكائن request يكون نسخة أخرى منحدرة من الكائن ServletRequest مناسبة لهذا البرتوكول

✶ الكائن **response:**

وهو مسئول عن حالة الاستجابة وينحدر من الكائن ServletResponse وفى معظم الحالات يكون هذا الكائن نسخة من الكائن HttpServletResponse هذا إذا كان البرتوكول HTTP هو المستخدم ويكون نسخة مختلفة إذا كان خلاف ذلك .

✶ الكائن **out:**

وهو مسئول عن إرسال الاستجابة إلى المستعرض ويكون نسخة منحدرة من الكائن JspWriter.

✶ الكائن **session:**

وهو نسخة من الكائن HttpSession لذلك لا يمكن استخدام هذا الكائن إلا في البرتوكول HTTP.

★ الكائن **pageContext:**

وهو نسخة من الكائن PageContext ومعظم المتغيرات الموجودة بالكائن pageContext موجودة بالكائن PageContext .

★ الكائن **config:**

ويمكنك من الوصول إلى الكثير من المعلومات حول اختيارات صفحة JSP ويكون نسخة من الكائن ServletConfig.

★ الكائن **page:**

يقدم هذا الكائن إمكانية مرجعية للوصول إلى الصفحة الحالية وهى نفس وظيفة العبارة this ولكن لأن البنية الهندسية للغة JSP تسمح باستخدام لغات متعددة فإن وجود هذا الكائن يستخدم في حالة استخدام لغة لا يوجد بها العبارة this.

★ الكائن **exception:**

يستخدم في حالة وجود خطأ في الصفحة ويمنحك هذا الكائن إمكانية للوصول إلى الصفحة الخاصة بالخطأ والحصول على معلومات منها.

★ الفئة **JspWriter:**

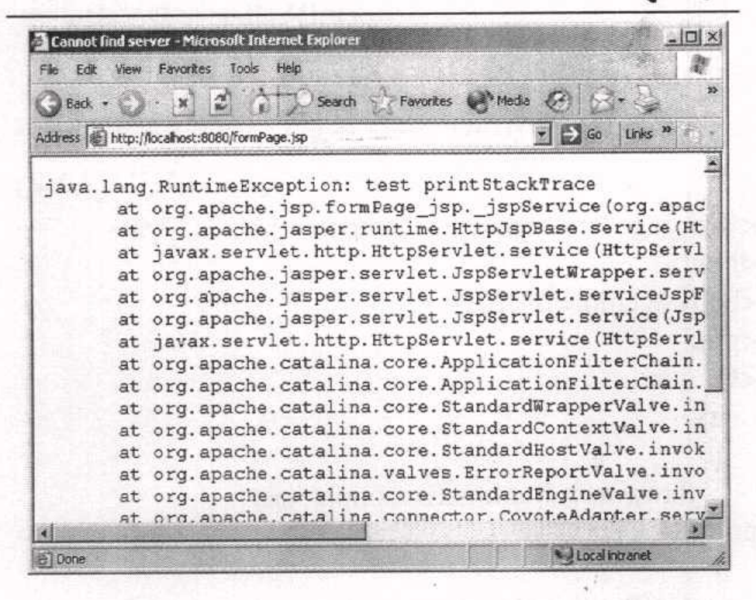
يشبه هذا الكائن PrintWriter إلا أنه مخصص أكثر للتعامل مع JSP والذاكرة المؤقتة ، فالوسيلتين print وprintln يقوما بإرسال خطأ عن طريق الكائن IOException في حالة ملء الذاكرة المؤقتة وهذا عكس نفس الوسائل بالكائن PrintWriter كما يعطيك إمكانيات أكثر للتحكم بهذه الذاكرة المؤقتة ولا داعي لأن تزعج نفسك بموضوع تخصيص ذاكرة مؤقتة للنصوص فيمكنك دائما استعمال الذاكرة الافتراضية .

★ استخدام الكائن **printWriter**:

يحتوى هذا الكائن على الوسيلة printStackTrace التي تقوم بطبع محتويات الذاكرة stack ولا يوجد بالكائن JspWriter مثيل لهذه الوسيلة وفيما يلي مثال لهذه الوسيلة

```
<%@ page language="java" import="java.io.*" %>
<html>
<body bgcolor="#ffffff">
<pre>
<%
  try
  {
    throw new RuntimeException("test
printStackTrace");
  }
  catch (RuntimeException exc)
  {
    PrintWriter pw = new PrintWriter(out);
    exc.printStackTrace(pw);
  }
%>
</pre>
</body>
</html>
```

ويكون نتيجة طباعة الذاكرة كما بالشكل التالي:



✶ التعامل مع الكائن pageContext:

يتعامل هذا الكائن كمخزن وسيط لتقديم المعلومات التي قد تحتاجها صفحة JSP ويتم إنشاء هذا الكائن عند بداية تحميل الصفحة وينتهي عند إغلاق أو إنهاء الصفحة ويتم إجراء تمهيد للمتغيرات المتضمنة بهذا الكائن في البداية فمثلا إذا كانت صفحة JSP تستخدم الكائن session فعادة يحتوى السيرفلت الخاص بها على هذا السطر

HttpSession session = pageContext.getSession();

كما يمكن عن طريق هذا الكائن الوصول إلى متغيرات وقيم حالات الطلب والاستجابة ، أيضا يمكن استخدامه للوصول إلى صفحات JSP أو سيرفلت أخرى ويتم تمهيد الكائنات المتضمنة بالكائن pageContext عن طريق الوسائل التالية:

```
public ServletRequest getRequest()
public ServletResponse getResponse()
public JspWriter getOut()
public HttpSession getSession()
public ServletContext getServletContext()
public ServletConfig getServletConfig()
public Object getPage()
public Throwable getException()
```

ويمكنك استخدام هذا الكائن في تخزين المتغيرات في نسخة من هذا الكائن ويكون لهذه الكائنات مدى محدود بالصفحة فلا يمكن الوصول إلى هذه المتغيرات من صفحة أخرى والوسائل المستخدمة لتخزين وقراءة وحذف المتغيرات في الكائن PageContext مشابهين لمثيلاتهم في الكائنات ServletRequest و HttpSession و ServletContext:

```
public Object getAttribute(String name)
public void setAttribute(String name, Object ob)
public void removeAttribute(String name)
```

والميزة الهامة هنا لاستخدام الكائن PageContext هو أننا نستطيع الوصول إلى أي متغير في أي مدى (صفحة أخرى مثلا) عن طريق إضافة معامل إلى الوسائل التالية:

```
Public Object getAttribute(String name, int scope)
Public void setAttribute(String name, Object object,
  int scope)
public void removeAttribute(String name, int scope)
```

ويمكن تحديد المعامل scope عن طريق استخدام احد الثوابت التالية :

PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE, APPLICATION_SCOPE

على سبيل المثال يمكن الوصول إلى كائن موجود في متغيرات session عن طريق استخدام العبارة التالية:

```
Object myObj = pageContext.getAttribute("ObjectIWant",
  PageContext.SESSION_SCOPE);
```

ويمكنك الوصول إلى أسماء كل المتغيرات في مدى معين عن طريق العبارة التالية:

```
public Enumeration getAttributeNamesInScope(int scope)
```

ويمكنك البحث في اى مدى عن كائن معين عن طريق العبارة التالية:

```
public Object findAttribute(String name)
```

يقوم الكائن pageContext بالبحث أولا في مدى الصفحة الحالية ثم في المدى المحدد في المعامل scope ثم أخيرا في المدى على مستوى البرنامج كله وخلال البحث يتم التوقف عند المدى الذي يجد فيه أولا ناتج البحث ولكن إذا كان هناك كائن موجود في المدى request ونفس الكائن في المدى application فإن العبارة findAttribute تقوم بإرجاع الكائن الذي يوجد في مدى request ، أما العبارة getAttributeScope فتقوم بإرجاع المدى الذي يوجد به كائن معين كما يتضح من العبارة التالية:

```
public int getAttributesScope(String name)
```

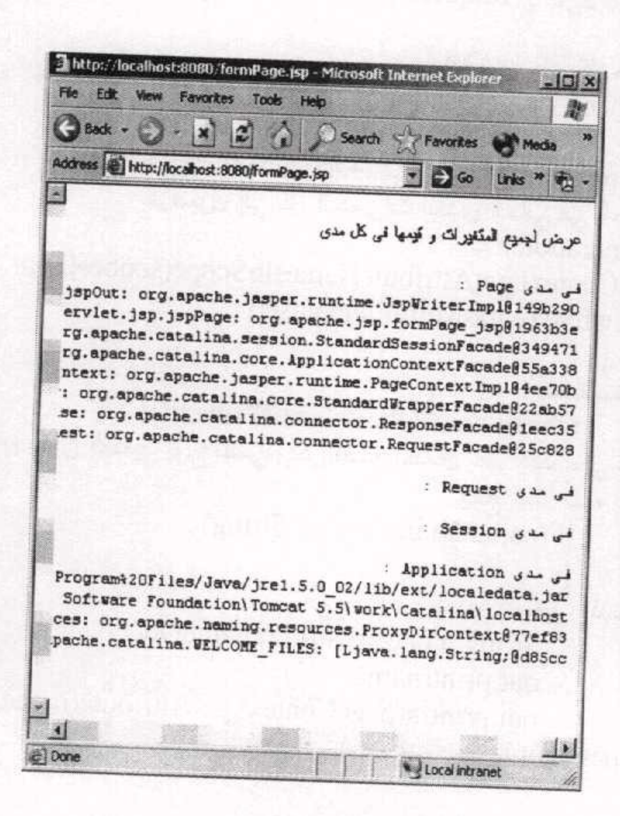
فمثلا إذا كان الكائن موجود في المدى request فإن العبارة السابقة تقوم بإرجاع القيمة PageContext.REQUEST_SCOPE

وفيما يلي مثال كامل لعرض كل متغير وقيمته في كل مدى في البرنامج:

```
<%@ page contentType="text/html; charset=windows-
1256" language="java" import="java.text.*,java.util.*"
%>
<html dir="rtl">
<body>
```

```
عرض لجميع المتغيرات وقيمها في كل مدى
<pre>
<%
// إنشاء مصفوفة لكل مدى محتمل
  int scopes[] = new int[] {
     PageContext.PAGE_SCOPE,
     PageContext.REQUEST_SCOPE,
     PageContext.SESSION_SCOPE,
     PageContext.APPLICATION_SCOPE };
// إنشاء أسم لكل مدى
  String scopeNames[] = new String[] {
     "Page", "Request", "Session", "Application"
  };
// حلقة تمر على كل مدى
  for (int i=0; i < scopes.length; i++)
  {
     out.println("في مدى"+scopeNames[i]+" :");
// الوصول إلى كل المتغيرات للمدى الحالي في الحلقة
Enumeration e =
pageContext.getAttributeNamesInScope(scopes[i]);
     while (e.hasMoreElements())
     {
// أسم المتغير
        Object nameOb = e.nextElement();
// يجب أن يكون المتغير نوع نصي ولكن نضع احتمال غير ذلك حتى لا
ينتج خطأ أثناء التنفيذ
        if (nameOb instanceof String)
        {
// طباعة المتغير وقيمته
           String name = (String) nameOb;
           out.print(name+": ");
           out.println(pageContext.getAttribute(name,
scopes[i]));
        }
```

```
        else
        {
            out.println("أسم المتغير ليس نصي بل: "+
              nameOb.getClass().getName());
        }
    }
    out.println();
  }
%>
</pre>
</body>
</html>
```



✶ **استخدام العبارة include وforward:**

عندما تقوم بتطوير برامج JSP قد تتكون الصفحة من عدة مقاطع مثل شريط عنوان وشريط للقائمة والجزء الأساسي للصفحة وشريط نهاية الصفحة في هذه الحالة يمكنك تضمين جميع ملفات JSP عن طريق العبارة <jsp:include> أو التمرير عن طريق العبارة <jsp:forward> وسنقوم لاحقا بتوضيح العبارتين.

أما إذا أردت استخدام نفس التقنية خارج نصوص JSP بلغة الجافا فيمكنك ذلك عن طريق ServletContext.getRequestDispatcher حتى تحصل على إمكانية الوصول للصفحة المطلوبة ثم بعد ذلك يمكنك تنفيذ عبارة التضمين include أو عبارة التمرير forward.

ويعطيك الكائن PageContext اختصارا للعبارتين فيمكنك فقط استخدام أحد العبارتين وتمرير اسم الصفحة المطلوبة كمعامل كما يلي:

```
public void forward(String url)
   throws IOException, ServletException
public void include(String url)
   throws IOException, ServletException
```

ويمكنك منع محتويات الذاكرة المؤقتة من الظهور في الصفحة قبل تنفيذ العبارة include عن طريق السطر التالي :

```
public void include(String url, boolean flush)
   throws IOException, ServletException
```

ووضع القيمة false للمعامل flush

★ الكائن JspEngineInfo:

أحيانا قد تريد بعض المعلومات عن الإصدار الحالي المستخدم مع محرك JSP وهذا يحدث أحيانا إذا قمت بتحديد طريقة عمل الصفحة مثلا إذا تم تنفيذها على محرك الإصدار 2.0 أو على الإصدار 1.2 بحيث تقوم بالاستفادة من كل مميزات الإصدار الحديث 2.0 ولا يتم تنفيذ بعض الخصائص بالصفحة إذا تم تنفيذها على الإصدار 1.2 ، وعن طريق معرفة رقم الإصدار يمكنك تحديد أي صفحة يتم استدعائها ويستخدم لذلك الكائن JspEngineInfo الذي يقوم بتعريف وسيله واحدة هي getSpecificationVersion التي تقوم باسترجاع رقم الإصدار كنص وتعرف كالتالي:

```
public String getSpecificationVersion()
```

وللحصول على رقم الإصدار يجب استخدام طريقه غير مباشرة عن طريق الكائن JspFactory الذي يستخدم داخليا بواسطة JSP والسيرفلت المتولدة عنها ويمكن الوصول إلى الكائن JspFactory عن طريق العبارة التالية:

```
JspFactory defaultFactory =
JspFactory.getDefaultFactory();
```

وبعد الوصول إلى الكائن JspFactory يمكنك استخدام الوسيلة getEngineInfo للحصول على رقم الإصدار كما يلي:

```
JspEngineInfo engineInfo =
defaultFactory.getEngineInfo();
```

وفيما يلي مثال يقوم بالحصول على رقم إصدار JSP وطبعة:

```
<%@ page contentType="text/html; charset=windows-
1256" language="java" import="java.text.*,java.util.*"
%>
```

```
<html dir="rtl">
<body>
الإصدار الحالي هو:
<%= JspFactory.getDefaultFactory().getEngineInfo().
  getSpecificationVersion() %>
</body>
</html>
```

وتكون النتيجة كما بالشكل:



الفصل السادس

عبارات التوجيه للغة JSP

فائدة عبارات التوجيه هي إعلام المزود (برنامج Tomcat) بمعلومات معينة حتى يمكنه تنفيذ البرنامج ، فمثلا إذا قمت باستخدام ملف خارجي فيمكنك توجيه المزود إلى مسار هذا الملف مثلا ويمكنك بالمثل تقديم الكثير من المعلومات للمزود كما سنرى الآن.

كيف يتم تعريف عبارات التوجيه:

تعلمنا فيما سبق أن كود الجافا يبدأ بعد العلامة <% وأن العلامة <%! تدل على وجود تعريف لوسائل أو خصائص كائن بعده والعلامة <%= تقوم بطبع نتيجة تعبير بلغة الجافا أما العلامة <%@ فهي المسئولة عن تعريف عبارة التوجيه ويوجد ثلاث عبارات توجيه في لغة JSP وهى:

- include
- page
- taglib

فتستخدم العبارة include لتضمين ملفات خارجية يحتاج إليها البرنامج وقت التنفيذ أما العبارة page فتستخدم كمحتوى لكل اختيارات وتضبيطات الصفحات الأخرى أما العبارة taglib فتستخدم لتوسيع لغة JSP فيمكنك استخدام وسوم (tags) إضافية من تعريفك غير مدرجة بلغة الجافا عن طريق وضعها في ملف مكتبة خارجي واستخدام هذه العبارة للتوجيه إلى هذا الملف.

★ العبارة include:

إذا كنت مبرمج لغة C++ فيمكنك التعرف سريعا على استخدام هذه العبارة التي تقوم بتضمين ملف خارجي للبرنامج ولا تقدم لغة الجافا مثيلا لها ولكن يمكن استخدام هذه العبارة في لغة JSP فقط.

ويقوم معالج JSP عند قراءة البرنامج بمعالجة الصفحة المتضمنة بعد عبارة include كما لو أنها جزء من الصفحة الحالية التي يقوم بمعالجتها وعلى سبيل المثال إذا أردت تضمين عنوان ثابت في كل الصفحات فيمكنك تخزين هذا العنوان في ملف وليكن title.htm ويمكنك تضمينه هكذا:

```
<%@ include file="title.html" %>
```

لاحظ أنك سوف تحتاج إلى ذكر المسار المتعلق بالصفحة الحالية للملف title.htm إذا كان يوجد بدليل فرعى آخر غير الحالي.

✶ العبارة page:

يتم وضع اختيارات الصفحة على الصيغة name=value فمثلا إذا أردت كتابة شرح للصفحة الحالية يمكن ذلك عن طريق السطر التالي:

```
<%@ page info="my new JSP page" %>
```

ويمكنك تضمين عبارة التوجيه page أكثر من مرة فيما عدا لو كان هناك عبارتين page تشتركان في نفس الاختيار ، وهذه القاعدة لها أستثنائين هما مع الاختيار import والاختيار pageEncoding .

✶ الاختيار contentType:

يمكنك عن طريق هذا الاختيار تحديد نوع بيانات الصفحة للمزود والنوع الافتراضي هوtext/html المستخدم مع صفحات HTML أما إذا كنت تريد طبع محتويات لغة XML فيجب تحديد التوجيه التالي:

```
<%@ page contentType="text/xml" %>
```

ويمكنك أيضا تحديد نوع الحروف المستخدمة في الصفحة فمثلا لإظهار حروف باللغة العربية في الصفحة بطريقه صحيحة يجب استخدام السطر التالي:

```
<%@ page contentType="text/html: charset=windows-1256 "%>
```

★ الاختيار pageEncoding:

وتستخدم لتعريف نظام تكويد الحروف الخاصة بلغة JSP ويكون النظام الافتراضي هو ISO-8859-1

★ الاختيار language:

يقوم بتحديد لغة النصوص المستخدمة في الصفحة وعادة ما تكون هي لغة الجافا ويتم تحديدها هكذا :

```
<%@ page language="java" %>
```

ويمكنك استخدام لغة أخرى مثل لغة JavaScript إذا قمت باستخدام محرك Resin الذي يقوم بتدعيمها فيتم تحديد اللغة المستخدمة كما يلي:

```
<%@ page language="javascript" %>
```

وفيما يلي مثال على استخدام لغة javascript لطبع العبارة الشهيرة : Hello World

```
<%@ page language="javascript" %>
<html>
<body>
<%
  var helloStr = 'Hello World!';
%>
<h1><%=helloStr%></h1>
</body>
</html>
```

لكن دائما في الإصدار 2.0 فإن لغة الجافا هي التي يمكن استخدامها.

✶ الاختيار EL is Ignored:

في الإصدار الثاني من لغة JSP تأتى هذه اللغة مع لغة ضمنية تسمى لغة التعبير يرمز لها EL والتي تجعل من الممكن إنشاء صفحات ديناميكية دون كتابة نصوص كود ويعطيك الاختيار EL is Ignored إمكانية تعطيل استخدام هذه اللغة في الصفحة .

✶ الاختيار Import:

سوف تقوم بكثرة باستخدام هذه العبارة التي تقوم باستيراد حزم مكتبات لغة الجافا الخارجية ، فمثلا لكي تتمكن من استخدام تعبيرات ومفردات خاصة بقواعد البيانات ولغة sql يجب أن تقوم باستيراد الحزمة java.sql.* كما يلي:

```
<%@ page import="java.sql.*" %>
```

وافتراضيا يقوم المزود باستيراد الحزم التالية:

java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*.

وكما ذكرنا يمكنك استخدام العبارة import أكثر من مرة كما يلي:

```
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
```

أو باستخدام نفس العبارة مع عدة قيم

```
<%@ page import="java.sql.*" import="java.util.*" %>
```

أما الأكثر احترافية هي وضع الحزم مع عبارة import واحدة كما يلي:

```
<%@ page import="java.sql.*,java.util.*" %>
```

ويقوم المعالج بترجمة الصفحة كلها قبل تحويلها إلى سيرفلت لذلك إذا قمت مثلا بوضع الاختيار import في آخر الصفحة فيتم تنفيذ الصفحة بصورة صحيحة بدون خطأ

```
<HTML>
<BODY>
<PRE>
<%
  out.println("Java is the best");
%>
<%@ page import="java.sql.*,java.util.*,java.math.*" %>
</PRE>
</BODY>
</HTML>
```

★ الاختيار info:

يمكنك عن طريق هذا الاختيار كتابة معلومات مفيدة عن وظيفة صفحة معينه ، وكمثال يمكنك كتابة الغرض من صفحة معينة هكذا:

```
<%@ page info="this page for new users" %>
```

ويمكنك الوصول إلى هذه المعلومات في الكود عن طريق الوسيلة Servlet.getServletInfo()

★ الاختيار session:

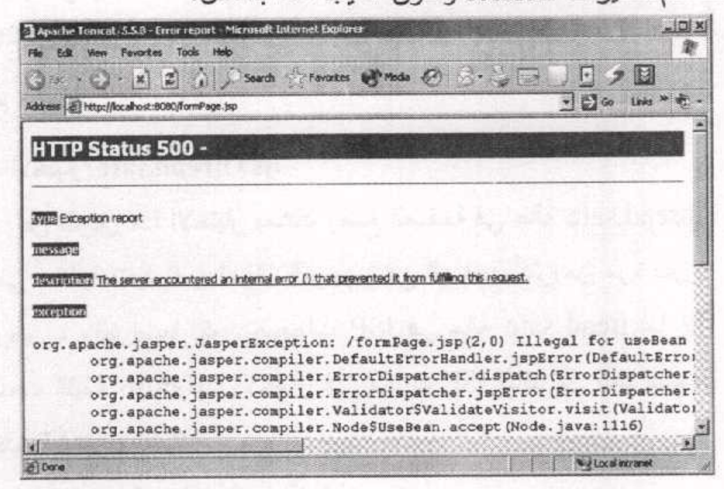
يمكنك عن طريق هذا الاختيار تشغيل أو إيطال استخدام المتغيرات المحفوظة session في الصفحة الحالية ويتم ذلك كما يلي:

```
<%@ page session="true" %>
```

والقيمة الافتراضية لهذه المتغيرات هي true مما يسمح بإمكانية استخدام المتغيرات والوصول إليها أما إذا أردت إيطال هذه المتغيرات فقم بوضع القيمة false والفائدة من إيطال المتغيرات المحفوظة session هي اختبار البرنامج إذا كنت من غير قصد تحاول قراءة متغير محفوظ ويمكن توضيح ذلك بالمثال التالي:

```
<%@ page session="false" %>
<jsp:useBean id="varQty" class="inventory.cart.customer"
scope="session"/>
<html>
<body>
</body>
</html>
```

يقوم هذا المثال بإنتاج خطأ وقت المعالجة compile time بسبب محاولة استخدام متغيرات session وتكون النتيجة كما بالشكل:



★ الاختيارين buffer وautoFlush:

يتحكم الاختيارين في الذاكرة المؤقتة buffer ويمكنك إبطال هذه الذاكرة نهائيا عن طريق وضع القيمة "none" للعبارة buffer كما يلي:

```
<%@ page buffer="none" %>
```

ويمكنك أيضا تحديد حجم buffer بمقياس الكيلوبايت فمثلا إذا أردت وضع الحجم 32 كيلوبايت يمكنك ذلك عن طريق أحد السطرين التاليين:

```
<%@ page buffer="32" %>
```

<%@ page buffer="32kb" %>

لاحظ أن حجم الـــ buffer هنا هو الحجم الأدنى الذي يمكن استخدامه ويستطيع محرك JSP أن يقوم بزيادته إذا دعت الحاجة .

ويقوم الاختيار autoFlush بتحديد إمكانية التخلص من الحجم الزائد عند امتلاء الذاكرة buffer أم لا والقيمة الافتراضية هي true فإذا امتلأت الذاكرة المؤقتة وحاولت أن تقوم بتخزين بيانات أخرى بها فإن محرك JSP يعطي رسالة خطأ وقت التنفيذ ليدل على امتلاء الذاكرة ، وعادة لا تحتاج لتغيير هذا الاختيار إلا في حالات قليله جدا حيث يفضل عدم تغيير حجم الذاكرة.

★ الاختيار **isThreadSafe**:

عن طريق هذا الاختيار يمكنك وضع الصفحة في حالة thread safe بمعنى إمكانية تنفيذ نفس الدوال الموجودة في الصفحة أكثر من مرة بدون تعارض ، وافتراضيا تكون صفحات JSP في حالة tread safe أما إذا وضعت القيمة false لهذا الاختيار فإن محرك JSP يتأكد من تنفيذ نسخة واحدة فقط من صفحتك .

وفيما يلي كيف نقوم بتغيير هذا الاختيار

<%@ page isThreadSafe="false" %>

لاحظ أن استخدام هذا الاختيار من الأشياء القديمة التي لم تعد مستعملة ويجب عدم استخدامه وهو موجود هنا فقط للتوافق مع الكود القديم.

★ الاختيار **errorPage** :

يمكنك عن طريق هذا الاختيار تحديد أي صفحة يتم إظهار معلومات الخطأ بها عند حدوث خطأ في الصفحة الحالية فعلى سبيل المثل يمكنك النداء على الصفحة "myErrorPage.jsp" عند حدوث خطأ هكذا:

```
<%@ page errorPage="myErrorPage.jsp"%>
```

✶ الاختيار isErrorPage:

يدل الاختيار على إمكانية استخدام الصفحة الحالية كصفحة خطأ لصفحة JSP أخرى فمثلا يمكن وضع القيمة true للاختيار في الصفحة "myErrorPage.jsp" ويمكن تنفيذ ذلك كما يلي:

```
<%@ page isErrorPage="true" %>
```

وبمجرد وضع القيمة true للاختيار يقوم معالج JSP بإنشاء الكائن exception ضمنيا الذي يتضمن الكائن Throwable المسئول عن تنفيذ صفحة الخطأ.

مثال:

سنقوم الآن بإنشاء ملفين الأول هو myErrorPage.jsp وهى الصفحة التي ستحتوى على رسالة الخطأ والملف الثاني هو exceptionPage.jsp الذي سنقوم فيه بتنفيذ خطأ أثناء التنفيذ وفيما يلي الكود للملفين:

الملف myErrorPage.jsp:

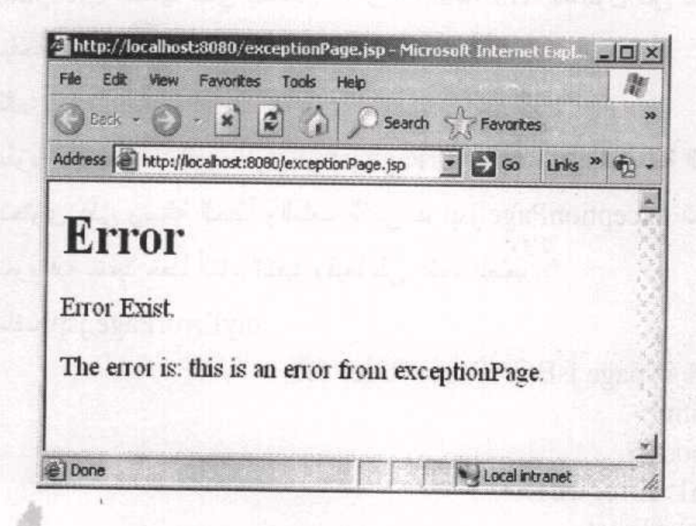
```
<%@ page isErrorPage="true" %>
<html>
<body>
<h1>Error</h1>
Error Exist.
<p>
The error is: <%= exception.getMessage() %>.
```

```
</body>
</html>
```

الملف exceptionPage:

```
<%@ page errorPage="myErrorPage.jsp" %>
<html>
<body>
This test you will not see because of an error
<%
   if (true) throw new RuntimeException("this is an error
from exceptionPage");
%>
</body>
</html>
```

ويكون تنفيذ الصفحة exceptionPage.jsp كما بالشكل:



- **الاختيار extends:**

يمكنك الاختيار extends من تحديد كائن أبوي لصفحة JSP وعادة لا تحتاج إلى تغيير الكائن الأبوي الذي يقوم بتحديده معالج JSP ولكن إذا كانت هناك ضرورة لإنشاء كائن أبوي فيمكنك ذلك.

ولاحظ أنك عند إنشاء كائن أبوي خاص بك يجب أن تراعى أن الكائن الأبوي يجب أن يعرف الواجهة JspPage وإذا كانت صفحات JSP الخاصة بك تستخدم البروتوكول HTTP والذي عادة يكون هو المستخدم فإن الكائن الأبوي يجب أن يتضمن تعريف HttpJspPage الذي يعتبر تطوير للواجهة JspPage.

ولأن الواجهة JspPage هي تطوير للواجهة Servlet فإن الكائن الأبوي يجب أن يقوم بتعريف كل الوسائل (methods) في واجهة Servlet ، هذا ويتضمن أيضا الوسائل init وservice وdestroy ، كما يجب تعريف وسائل الواجهة Servlet في الكائن الأبوي بدون إمكانية للتعديل في هذه الوسائل وهى التقنية التي تسمى (Override).

وعادة إذا قمت بإنشاء الكائن الأبوي بحيث يقوم بتطوير HttpServlet والذي يقوم بمعظم المجهود لك سوف تحتاج فقط إلى تعريف وسيلتين هما:

```
public void jspInit()
public void jspDestroy()
```

ويجب أن تقوم بالنداء على الوسيلة jspInit من ضمن الوسيلة init الموجودة بالكائن الأبوي والوسيلة jspDestroy من خلال الوسيلة destroy كما يجب أن تقوم الوسيلة service بتنفيذ الوسيلة

jspService_ وهذه الوسيلة معرفه بالفعل في الواجهة HttpJspPage ومعلن عنها هكذا:

```
public void _jspService(HttpServletRequest request,
   pttpServletResponse response)
   throws ServletException, IOException
```

وفيما يلي مثال على إنشاء كائن أبوي لصفحة JSP:

```
package myjsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
// superclass كائن ابوى
public abstract class JSPSuperclass extends HttpServlet
   implements HttpJspPage
{
// تعريف الوسيلة init بدون إمكانية تعديل
   public final void init(ServletConfig config)
      throws ServletException
   {
//النداء على الوسيلة init للكائن الابوى
      super.init(config);
//JSP تمهيد صفحة
      jspInit();
   }
// النداء على الوسيلة destroy بدون إمكانية تعديل
   public final void destroy()
   {
      super.destroy();
      jspDestroy();
```

```
  }
  public final ServletConfig getServletConfig()
  {
    return super.getServletConfig();
  }
//service تعريف الوسيلة
  public final void service(ServletRequest request,
     ServletResponse response)
     throws ServletException, java.io.IOException
  {
     super.service(request, response);
  }
  public final void service(HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, java.io.IOException
  {
// سنقوم بوضع أي بيانات هنا لتوضيح طريقة عمل الكائن الابوى
request.setAttribute("Hello", "Hello from superclass");
//قم بالنداء على الوسيلة jspService_ لتنفيذ الصفحة//
     _jspService(request, response);
  }
//jspInit تعريف الوسيلة
  public void jspInit()
  {
  }


//jspDestroy تعريف الوسيلة
  public void jspDestroy()
  {
```

```
    }
// الوسيلة jspService_ تم تعريفها في السيرفلت الناشئ عن الصفحة
    public abstract void _jspService(HttpServletRequest
request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException;
}
```

وفيما يلي الكود الخاص بالصفحة التي تقوم باستخدام الكائن الابوى

```
<%@ page extends="myjsp.JSPSuperclass"%>
<html>
<body>
This is a subclass of a custom servlet.
Parent class message is:
<%= request.getAttribute("Hello") %>
</body>
</html>
```

★ العبارة taglib:

يمكنك عن طريق العبارة taglib إنشاء وسوم معدله مثل وسوم HTML أوXML وعن طريق استخدام هذه الوسوم المعدلة يمكنك تقليل حجم كود الجافا المستخدم في الصفحة .

ويمكن استخدام العبارة taglib لتحميل مكتبة وسوم ولكن يجب أن تحدد المسار URI والاسم المبدئي هكذا :

```
<%@ taglib
uri="http://myjsp.egyptbooks.net/taglib/testlib"
 prefix="newtag" %>
```

ويمكنك استخدام الوسوم المعدلة على الشكل <prefix:tag_name> حيث أن المعامل prefix هو نفسه الذي تقوم بتحديده في العبارة الموجهة taglib والمعامل tag_name هو أسم الوسم المعرف في مكتبة الوسوم .

على سبيل المثال إذا كان هناك وسم اسمه "newstreet" فيمكنك أستخدام الوسم مع الاسم المبدئي "newtag" هكذا:

<newtag:newstreet>

الفصل السابع

أهم المهارات للغة JSP

في هذا الفصل ستتعلم كيف تقوم بتضمين ملفات الموارد (صور أو بيانات) في صفحات JSP أو السيرفلت ، وأيضا كيف تقوم بنقل الزائر من صفحة لأخرى تلقائيا وكيف تقوم بإنشاء عناصر قابلة للاستخدام مرات عديدة وأخير كيف تقوم باستخدام applet في الصفحة.

وستتعلم أيضا كيف نقوم بتنظيم الكود الخاص بالجافا مع نصوص HTML حيث أن أكثر من أكثر عيوب صفحات JSP أنها تكون صعبة الفهم والتعديل عندما تكون كبيرة الحجم .

تضمين موارد البرنامج:

يمكنك استخدام الوسوم المعدلة (tags) لجعل البرنامج أكثر سهوله في الفهم والتعديل والتنسيق بين كود الجافا وعناصر لغة JSP بحيث تستغني تمام عن كتابه نصوص أو تعبيرات غير منظمه .

وعلى سبيل المثال فإن تقسيم الكود الخاص بالصفحة من أهم وسائل التنظيم ، فعادة يكون لبرنامج الويب نفس العنوان في كل الصفحات ، ويمكنك إنشاء صفحة مخصوصة للعنوان وتضمينها في كل الصفحات

ويمكنك أيضا استخدام تقنية التضمين إما خلال وقت المعالجة أو وقت التنفيذ وتتميز عملية التضمين وقت المعالجة بأنها أسرع في التنفيذ ، أما بالنسبة للسيرفلت فيمكنك تضمين الملفات فقط وقت التنفيذ ، ولتضمين ملف وقت المعالجة يمكنك استخدام العبارة include كما يلي:

```
<%@ include file="file_to_include" flush="true"%>
```

مثال:

سنقوم الآن بتضمين ملف ثابت وهو العنوان في ملف آخر ويمكنك رؤية كود الملف الآخر موجود في السيرفلت الناشئ عن الصفحة.

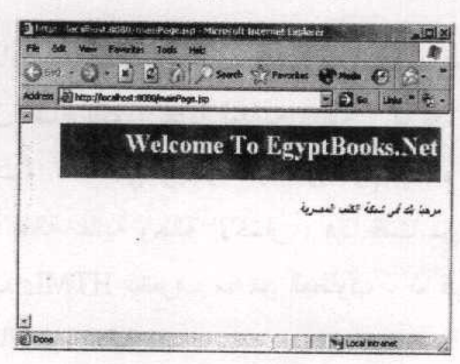
ملف العنوان titlePage.htm:

```
<table width="484" height="65" bgcolor="#0099FF">
<tr>
  <td width="476" height="61"><h1><font
color="#FFFFFF">Welcome To
EgyptBooks.Net</font></h1></td>
  </tr>
</table>
```

ملف الصفحة الاساسية mainPage.jsp:

```
<%@ page contentType="text/html; charset=windows-
1256" %>
<html dir="rtl">
<body>
<%@ include file="titlePage.htm" %>
<p><em><strong> مرحبا بك في شبكة الكتب المصرية
</strong>
</em>
<p>
</body>
</html>
```

وتكون نتيجة تنفيذ ذلك على الشكل التالي:



كما يمكنك تضمين ملفات أخرى في الملف "titlePage.htm" ، وإذا قمت باستخدام أي متغيرات أو وسائل في الملف الرئيسي يمكنك أيضا الوصول إلى هذه المتغيرات والوسائل في الملف المتضمن .

لاحظ أنه إذا قمت باستخدام مزود أخر غير مزود Tomcat فإنه عند تغيير الملفات المتضمنة فإن المزود لا يعرف ما إذا كانت الملفات تحتاج لإعادة معالجة أم لا ، لذلك يجب عليك أن تتأكد من إعادة عملية المعالجة.

تضمين الملفات أثناء وقت التنفيذ:

تتميز هذه الطريقة بالمرونة لدمج سيرفلت وصفحات JSP معا فإذا أردت دمج سيرفلت أو صفحة JSP لصفحة JSP معينة قم باستخدام العبارة الموجهة <jsp:include> كما يلي:

```
<jsp:include page="file_to_include" flush="true"/>
```

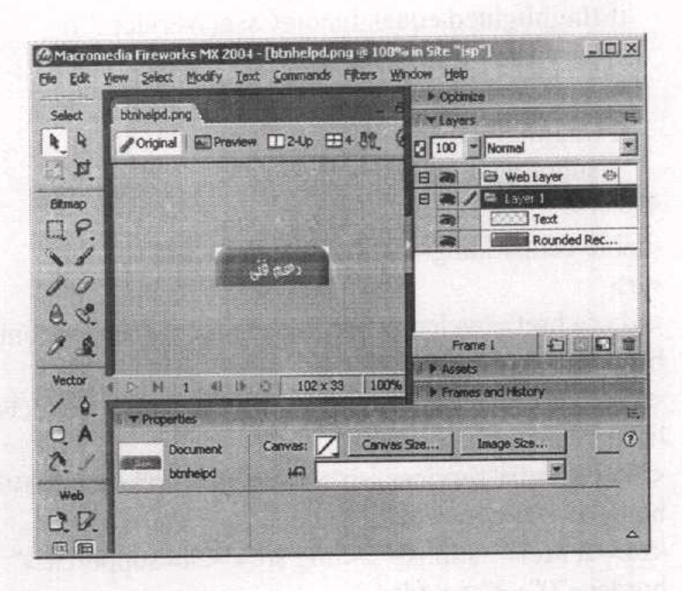
مثال:

سنقوم الآن بإنشاء برنامج ويب بسيط عبارة عن ملف للعنوان ثابت وهو"titlePage.htm" من المثال السابق والملف "menu.jsp" وهو يحتوى على قائمة للتنقل في الموقع وصفحة لكل مفتاح في القائمة وفيما يلي الخطوات:

- قم بإنشاء أربع مفاتيح بأي برنامج محرر للرسوم وقم بوضع اسم على كل مفتاح ليكون عندنا المفاتيح التالية : "مرحبا"،"المنتجات"، "الخدمات"، "دعم فني" ويجب عليك أن تقوم بإنشاء المفاتيح في حالتين: حالة عادية وحالة الاختيار ، فإذا قمت من قبل بإنشاء صفحات HTML ستعرف ما هو المطلوب ، ثم قم بتسمية كل مفتاح بالأسماء التالية :

"btnWel.png","btnProduct.png","btnServ.png",
"btnhelp.png"

ولقد سميت هنا المفاتيح في حالة الاختيار بنفس الأسماء مع إضافة حرف "d" لكل منها



- قم الآن بفتح ملف جديد وقم بتسميته "menu.jsp" واكتب الكود التالي به:

```
<%
// See which menu item should be highlighted.
   String highlighted =
request.getParameter("highlighted");
// Set the names for the individual menu items.
   String welcome = "btnWel.png";
   if (highlighted.equalsIgnoreCase("welcome"))
```

```
    welcome = "btnWeld.png";
  String products = "btnProduct.png";
  if (highlighted.equalsIgnoreCase("products"))
    products = "btnProductd.png";
  String services = "btnServ.png";
  if (highlighted.equalsIgnoreCase("services"))
    services = "btnServd.png";
  String support = "btnhelp.png";
  if (highlighted.equalsIgnoreCase("support"))
    support = "btnhelpd.png";
%>
<table cellpadding="0" cellspacing="0">
<tr>
<td><a href="welcome.jsp"><img src="<%=welcome%>"
border="0"></a></td>
<td><a href="products.jsp"><img src="<%=products%>"
border="0"></a></td>
<td><a href="services.jsp"><img src="<%=services%>"
border="0"></a></td>
<td><a href="help.jsp"><img src="<%=support%>"
border="0"></a></td>
</table>
```

- قم بالحفظ وإنشاء ملف جديد لكل مفتاح سنقوم هنا فقط بإنشاء ملف الدعم الفني وعليك تكملة المثال:

الملف أسمه "help.jsp":

```
<%@ page contentType="text/html; charset=windows-
1256" %>
<html dir="rtl">
<body bgcolor="#ffffff">
<%@ include file="titlePage.htm"%>
```

```
<jsp:include page="menu.jsp" flush="true">
  <jsp:param name="highlighted" value="support"/>
</jsp:include>
<p>
<h1>الدعم الفني</h1>
<p>
<strong><i>يمكنك كتابة أي نص هنا</i>
<br>
</strong><i>يمكنك كتابة أي نص هنا</i>
<i>يمكنك كتابة أي نص هنا</i> <i>يمكنك كتابة أي نص هنا</i>
<p>
<strong><i>يمكنك كتابة أي نص هنا</i><br>
</strong><em>يمكنك كتابة أي نص هنا يمكنك كتابة أي نص هنا
يمكنك كتابة أي نص هنا يمكنك كتابة أي نص هنا</em>.
</body>
</html>
```

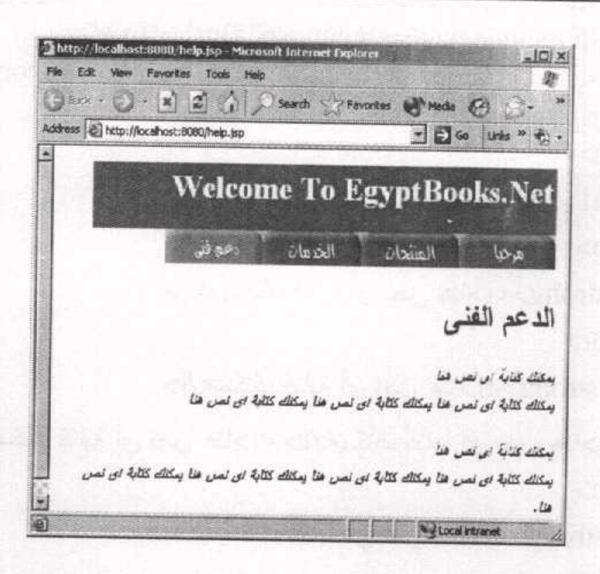
لاحظ انه يجب إنشاء الملفات التالية بالأسماء كما تم تحديدها في الكود:

welcome.jsp, products.jsp, services.jsp, help.jsp

ولا تنسى أيضا تغيير السطر التالي لكل صفحة

```
<jsp:param name="highlighted" value="support"/>
```

ويكون نتيجة تنفيذ المثال كما بالشكل:



ملاحظات: لا يمكن استخدام أي عناصر عنوان في البيانات المرسلة URL في الملف المتضمن مثل الكوكيز

لاحظ تمرير قيم المتغيرات عن طريق العبارة :

```
<jsp:param name="highlighted" value="support"/>
```

فيمكن للملف المتضمن أن يصل لجميع المتغيرات في الصفحة الرئيسية لأن له إمكانية الوصول إلى الكائن request ويتم ذلك بالصيغة التالية:

```
<jsp:include page="page_to_include" flush="true">
   <jsp:param name="my_Param_Name"
value="Param_Value"/>
</jsp:include>
```

لاحظ أن العبارة الموجهة <jsp:include> يتم إقفالها بالوسم </ طبقا للشكل القياسي المستعمل مع بيانات XML ويمكنك أيضا إقفاله بالوسم <jsp:include/>.

يستخدم الملف المتضمن الوسيلة request.getParameter والوسيلة request.getParameterValues للوصول إلى المعاملات وقيمها وإذا قمت باستخدام الوسيلة request.getParameter تحصل على المعامل من الوسم <jsp:param> أما إذا استخدمت الوسيلة request.getParameterValues فسوف تحصل على القيم من الوسم <jsp:param> وأيضا القيم الممررة من المستعرض. ولاحظ أن المعاملات الممررة عن طريق الوسم <jsp:param> ظاهرة للملف المتضمن فقط وليس للملف الأصلي .

مثال:

سنقوم في هذا المثال بتوضيح عملية تمرير متغير معين إلى الملف المتضمن

الملف mainPage.jsp:

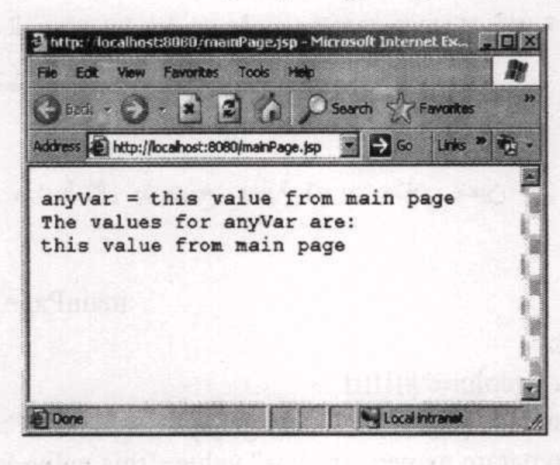
```
<html>
<body bgcolor="#ffffff">
<jsp:include page="includePage.jsp" flush="true">
   <jsp:param name="anyVar" value="this value from
main page"/>
</jsp:include>
</body>
</html>
```

الملف includePage.jsp:

```
<pre>
```

```
<%
   String anyVar = request.getParameter("anyVar");
   String anyVars[] =
request.getParameterValues("anyVar");
   out.println("anyVar = "+ anyVar);
   out.println("The values for anyVar are:");
   for (int i=0; i < anyVars.length; i++)
   {
      out.println(anyVars[i]);
   }
%>
</pre>
```

وتكون نتيجة تنفيذ المثال السابق كما بالشكل:



لاحظ هنا الفرق بين استخدام الوسيلة request.getParameter والوسيلة request.getParameterValues ويمكننا توضيح ذلك أكثر إذا قمت بكتابة العنوان URL للمثال كما يلي:

http://localhost:8080/mainPage.jsp?anyVar="this is a new value from browser"

والسؤال الآن كيف يمكن تضمين الملفات مع السيرفلت ؟

يمكنك ذلك عن طريق دوال API ولكن ليس بطريقه مباشرة للأسف فيجب أن تحصل على ما يسمى الكائن المرسل (request dispatcher) للملف الذي تريد تضمينه . ولكن لحسن الحظ هناك طريقه سريعة لأداء ذلك وهى استخدام الوسيلة request.getRequestDispatcher بحيث يكون المعامل هو مسار URL الملف الذي تريد تضمينه

```
RequestDispatcher d =
     request.getRequestDispatcher("url_of _to_include");
d.include(request, response);
```

مثال:

الملف mainServlet.java:

```
package examples;
import javax.servlet.*;
import java.io.*;
public class MainFormServlet extends GenericServlet
{
  public void service(ServletRequest request,
     ServletResponse response)
     throws IOException, ServletException
  {
//HTML السطر التالي يؤكد للمزود أن الاستجابة هي من نصوص
     response.setContentType("text/html");
//PrintWriter لكتابة المخرجات يجب أولا الحصول على الكائن
     PrintWriter out = response.getWriter();
```

```
//HTML كتابة المخرجات في صورة نصوص
        out.println("<html>");
        out.println("<body>");
//request dispatcher الحصول على الكائن المرسل
        RequestDispatcher dispatcher =
            request.getRequestDispatcher(
                "/includPage.jsp");
        dispatcher.include(request, response);
        out.println("</body>");
        out.println("</html>");
    }
}
```

- لاحظ أنه يمكنك استخدام الكائن GenericServlet أو الكائن HttpServlet ولكننا استخدمنا GenericServlet لأنه يحتاج فقط إلى الوسيلة service.
- لا تنسى كتابة قيمة المعامل في سطر المسار URL عند تجربة المثال.
- إذا أردت تمرير معامل للملف المتضمن للسيرفلت فيمكن أداء ذلك هكذا:

```
getRequestDispatcher("myPage.jsp?myparam=anyvalue");
```

تقنية التمرير لصفحات أخرى:

يمكنك تمرير حالة الطلب request إلى صفحة أخرى أو سيرفلت بحيث لا تكون الصفحة المرسلة مسئولة عن معالجة الطلب ، وأكثر مثال لهذا الاستخدام هو التمرير إلى صفحة أخرى لمعالجة خطأ ، ومثال آخر هو الحاجة للاستجابة بطرق مختلفة حسب حالة الطلب وفى هذه الحالة

سوف تقوم صفحة JSP أو السيرفلت بتحليل الطلب وتحديد أي صفحة يتم التمرير إليها .

★ طريقة التمرير من صفحة JSP:

صيغة العبارة الخاصة بالتمرير <jsp:forward> مشابه لحد كبير لصيغة عبارة التضمين وفيما يلي مثال على ذلك:

```
<jsp:forward page="page_to_forward"/>
```

لاحظ أنه إذا حدث خطأ IllegalStateException عند التمرير فهذا معناه أنه لا توجد ذاكره مؤقتة buffer ويجب تشغيلها كما درسنا عن طريق عبارات التوجيه.

مثال:

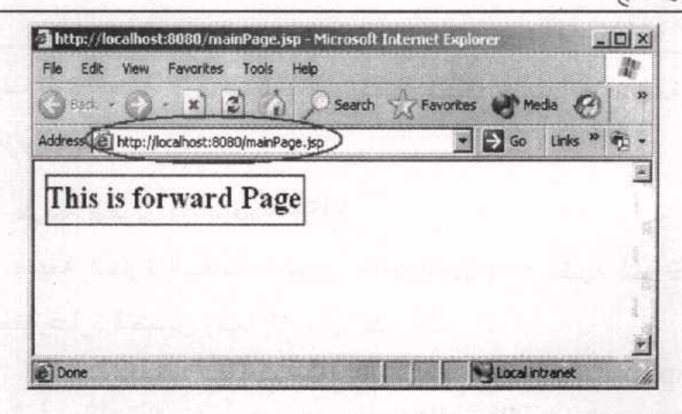
الملف mainPage.jsp:

```
<html>
<body>
you will not see this text
<jsp:forward page="ForwardedPage.jsp"/>
</body>
</html>
```

- قم الآن بفتح ملف جديد ForwardedPage.jsp:

```
<html>
<body>
<h3>This is forward Page
</h3>
</body>
</html>
```

- وعند التنفيذ ستجد أن الصفحة الأولى mainPage.jsp قد مررت الطلب إلى الصفحة الثانية ForwardedPage.jsp ويكون الشكل كما يلي:



✱ التمرير من سيرفلت:

ستقوم هنا أيضا باستخدام الكائن RequestDispatcher ولكن بدلا من النداء على include ستقوم باستخدام الوسيلة forward كما يلي:

```
RequestDispatcher d =
    request.getRequestDispatcher("URL");
d.forward(request, response);
```

مثال:

سنقوم الآن بكتابة سيرفلت يقوم بالتمرير إلى نفس الصفحة ForwardedPage.jsp المستخدمة في المثال السابق حيث يتم حذف مخرجات السيرفلت قبل حدوث التمرير:

الملف mainServlet.java:

```
package myServlets;
import javax.servlet.*;
import java.io.*;
public class mainServlet extends GenericServlet
{
  public void service(ServletRequest request,
    ServletResponse response)
```

```
        throws IOException, ServletException
    {
//HTML تحديد نوع الاستجابة
        response.setContentType("text/html");
//PrintWriter كتابة المخرجات عن طريق الكائن
        PrintWriter out = response.getWriter();
//HTML كتابة نصوص للمستعرض
        out.println("<html>");
        out.println("<body>");
        out.println("You will not see this text");
//الوصول إلى الكائن RequestDispatcher لتنفيذ التمرير
        RequestDispatcher dispatcher =
            request.getRequestDispatcher(
                "/ForwardedPage.jsp");
        dispatcher.forward(request, response);
        out.println("</body>");
        out.println("</html>");
    }
}
```

- يمكنك تمرير المعاملات بالنسبة لتقنية التمرير تماما مثل تقنية التضمين include باستخدام الوسم <jsp:param>

★ تمرير كائنات بين صفحات JSP والسيرفلت:

أحيانا يكون هناك كمية ضخمة من البيانات تحتاج إلى تمريرها بين سيرفلت وJSP ، ولقد عرفت كيف يمكن أداء ذلك عن طريق المعاملات النصية ولكن سيكون من الصعب والغير منطقي تحويل هذه الكمية الكبيرة التي قد تزيد عن 20 معامل كل في نص منفصل ولكن يمكنك تمرير دفعة بيانات مرة واحدة بطريقة منظمة وهى عن طريق تمرير كائن من كائنات جافا مرة واحدة.

يمكن تنفيذ ذلك عن طريق تقنية المتغيرات المحفوظة session التي يمكنك عن طريقها تخزين كائن بواسطة الوسيلتين getAttribute وsetAttribute ولكن عيب هذه الطريقة أن الكائن سيظل مخزنا حتى تقوم بكتابة كود مخصوص لحذف الكائن من session بعد استعماله.

ولكن الطريقة الأنسب هي استخدام الكائن request واستخدام الوسائل getAttribute وsetAttribute وremoveAttribute تماما مثل session وتعريف هذه الوسائل هو:

```
public void setAttribute(String name, Object value)
public Object getAttribute(String name)
public void removeAttribute(String name, Object value)
```

ونظرا لأن الكائن request لا يكون موجودا بعد انتهاء الطلب فإنك لا تحتاج إلى كتابة كود مخصوص لإزالة الكائن فإذا لم ترسل استجابة إلى المستعرض فإن الكائن request يزول وإذا حدث طلب أخر يتم أنشاء request جديد.

✱ تقسيم البرنامج لملفات منظمة:

في الحياة العملية سوف تقوم بكتابة برامج كبيرة الحجم فإذا قمت بكتابة كل الكود في ملف واحد سينتهي بك الأمر إلى كود غير قابل للفهم والتعديل لذلك يتم استخدام الوسم <jsp:include> لتنظيم الكود في ملفات صغيرة.

من ضمن الفوائد الكبيرة والمهمة هي تقسيم الكود لإنشاء ملفات يمكن إعادة استخدامها ، فمثلا إذا أردت إظهار مصفوفة في عدة صفحات في برنامجك فلا يكون من المنطقي تكرار الكود في كل صفحة تحتاج فيها إلى إظهار هذه المصفوفة .

مثال:

فيما يلي مثال لسيرفلت يقوم بالحصول على البيانات من مصفوفة أو كائن معين تم تمريره كمعامل ويقوم بإظهار هذه المصفوفة ويتم ذلك عن طريق استخدام الوسيلة request.getAttribute للحصول على البيانات والوسيلة request.getParameter للحصول على المعاملات:

الملف TableServlet:

```
import javax.servlet.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
public class TableServlet extends GenericServlet
{    public static final Class[] NO_PARAMS = new
Class[0];
   public void service(ServletRequest request,
ServletResponse response)
       throws IOException, ServletException
```

```
{
//TABLE, TR, TD, TH الحصول على معاملات الجدول//
    String tableOptions =
request.getParameter("tableOptions");
    if (tableOptions == null) tableOptions = "";
    String trOptions = request.getParameter("trOptions");
    if (trOptions == null) trOptions = "";
    String tdOptions =
request.getParameter("tdOptions");
    if (tdOptions == null) tdOptions = "";
    String thOptions =
request.getParameter("thOptions");
    if (thOptions == null) thOptions = "";
//الحصول على أسم الكائن الذي يحتوى البيانات//
    String data = request.getParameter("data");
    if (data == null)
    {
      getServletContext().log("No data available");
      throw new ServletException(
         "No data parameter available");
    }
// الحصول على البيانات نفسها //
    Object dataOb = request.getAttribute(data);
    if (dataOb == null)
    {
      getServletContext().log("No data object found");
      throw new ServletException(
         "Can't locate the data object named "+
         data);
    }
```

```
الحصول على أسماء الحقول والوسائل لعرضها في كل عمود //
        String[] columns =
request.getParameterValues("column");
الحصول على نوع كل حقل //
        String[] columnType =
request.getParameterValues("columnType");
الحصول على عناوين الأعمدة //
        String[] columnHeaders =
request.getParameterValues(
        "columnHeader");
إنشاء الجدول الذي يتم فيه عرض البيانات //
        Hashtable columnAccessors =
            getAccessors(dataOb, columns);
أولا يتم طباعة عناوين الأعمدة //
        PrintWriter out = response.getWriter();
        out.println("<table "+ tableOptions+">");
اختبار إذا كان هناك عناوين للأعمدة يتم طباعتها كلها //
        if (columnHeaders != null)
        {
            out.println("<tr "+trOptions+">");
            for (int i=0; i < columnHeaders.length; i++)
            {
                out.print("<th "+thOptions+">");
                out.println(columnHeaders[i]);
                out.println("</th>");
            }
            out.println("</tr>");
        }
```

اختبار إذا كان الكائن عبارة عن مصفوفة يتم الحصول على معاملاتها عن طريق //

حلقه

تكراريه//

```
    if (dataOb instanceof Vector)
    {
      Vector v = (Vector) dataOb;
      Enumeration e = v.elements();

      while (e.hasMoreElements())
      {
// For each row, print out the <tr> tag plus any options.
//يتم طباعة الوسم <tr> لكل صف مع أي متغيرات قد توجد//
        out.println("<tr "+trOptions+">");
// طباعة قيم كل عمود //
        printRow(out, e.nextElement(),
          columns, columnType,
          columnAccessors, tdOptions);
        out.println("</tr>");
      }
    }
```

إذا كان الكائن مصفوفة يتم الدخول في حلقه تكراريه للوصول لكل كائن //

به بيانات

```
    else if (dataOb instanceof Object[])
    {
      Object[] obs = (Object[]) dataOb;
      for (int i=0; i < obs.length; i++)
      {
//طباعة الوسم <tr> لكل صف بالإضافة لكل المتغيرات//
```

```
            out.println("<tr "+trOptions+">");
// طباعة قيم كل عمود لكل صف في الحلقة
            printRow(out, obs[i],
              columns, columnType,
              columnAccessors, tdOptions);
            out.println("</tr>");
          }
      }
      out.println("</table>");
   }
   protected void printRow(PrintWriter out, Object ob,
      String[] columns, String[] columnTypes,
      Hashtable columnAccessors, String tdOptions)
      throws ServletException
   {
// حلقه تكراريه لكل الأعمدة
      for (int i=0; i < columns.length; i++)
      {
// الحصول على قيمة العمود الحالي في الحلقة
         Object value = getColumnValue(ob, columns[i],
            columnAccessors);
//<td>  طباعة الوسم
         out.print("<td "+tdOptions+">");
//إذا كان نوع البيانات data يتم طباعتها//
         if (columnTypes[i].equalsIgnoreCase("data"))
         {
            out.print(value);
         }
//<img> إذا كان نوع البيانات صوره يتم طباعة الوسم//
```

```
        else if
(coiumnTypes[i].equalsIgnoreCase("image"))
        {
          out.print("<img src=\""+value+"\">");
        }
        out.print("</td>");
      }
    }
//يمكنك استخدام الحقل أو وسيله لإحضار القيمة من الكائن
  protected Object getColumnValue(Object ob, String
columnName,
      Hashtable columnAccessors)
      throws ServletException
  {
//الحصول على الكائن المستخدم للحصول على قيمة العمود
      Object accessor =
columnAccessors.get(columnName);
//إذا كان العمود حقل
      if (accessor instanceof Field)
      {
//يتم استخدام الوسيلة get للحصول على القيمة
        try
        {
          Field f = (Field) accessor;
          return f.get(ob);
        }
//كود للتعامل مع الخطأ الخاص بالا عمده
        catch (IllegalAccessException exc)
        {
          getServletContext().log(
```

```
            "Error getting column "+
            columnName, exc);
         throw new ServletException(
            "Illegal access exception for column "+
            columnName);
      }
   }
// إذا كان العمود وسيله
   else if (accessor instanceof Method)
   {
// يتم استدعاء هذه الوسيلة وتنفيذها
      try
      {
         Method m = (Method) accessor;
// القيمة NO_PARAMS هي مصفوفة خاليه
         return m.invoke(ob, NO_PARAMS);
      }
//كود للتعامل مع أي خطأ في استدعاء الوسيلة
      catch (IllegalAccessException exc)
      {
         getServletContext().log(
            "Error getting column "+
            columnName, exc);
         throw new ServletException(
            "Illegal access exception for column "+
            columnName);
      }
      catch (InvocationTargetException exc)
      {
         getServletContext().log(
```

```
            "Error getting column "+
            columnName, exc);
          throw new ServletException(
            "Invocation target exception "+
            "for column "+columnName);
        }
    }
//إذا كان العمود ليس حقل أو وسيله يتم إرجاع القيمة null//
    return null;
  }
//  إنشاء جدول معرف به أسماء الأعمدة //
  protected Hashtable getAccessors(Object ob, String[]
columns)
    throws ServletException
  {
    Hashtable result = new Hashtable();
// First, get the Class for the kind of object being
displayed.
//أولا قم بالحصول على نوع الفئة الخاص بالكائن الذي يتم عرضه//
    Class obClass = null;
    if (ob instanceof Object[])
    {
//إذا كان الكائن عبارة عن مصفوفة يتم الحصول على أول كائن فيها//
      Object[] obs = (Object[]) ob;
//إذا كان لا يوجد كائنات فلا تقوم بعملية ملء الجدول//
      if (obs.length == 0) return result;
      obClass = obs[0].getClass();
    }
    else if (ob instanceof Vector)
    {
```

```
إذا كان الكائن مصفوفة متجهه فنقوم بالحصول على أول عنصر فيها//
        Vector v = (Vector) ob;
إذا كان لا يوجد كائنات فلا تقوم بعملية ملء الجدول//
        if (v.size() == 0) return result;
        obClass = v.elementAt(0).getClass();
    }
حلقه تكراريه للأعمدة للوصول إلى حقل أو وسيلة كل عمود//
    for (int i=0; i < columns.length; i++)
    {
التأكد من وجود حقل يماثل أسم العمود//
        try
        {
            Field f = obClass.getField(columns[i]);
إذا وجدت الحقل قم بوضعه في المكان الصحيح وانتقل للاسم التالي//
            result.put(columns[i], f);
            continue;
        }
        catch (Exception ignore)
        {
        }

الآن قم بالتأكد من وجود وسيله لاسم العمود//
        try
        {
القيمة NO_PARAMS هي مصفوفة خاليه//
            Method m = obClass.getMethod(columns[i],
                NO_PARAMS);
        result.put(columns[i], m);
```

```
        }
        catch (Exception exc)
        {
          getServletContext().log(
            "Exception location field "+
            columns[i], exc);
          throw new ServletException(
            "Can't locate field/method for "+
            columns[i]);
        }
      }
      return result;
    }
}
```

قم الآن بفتح ملف جديد "viewtable.jsp" واكتب الكود التالي

```
<html>
<body>
<%
تمهيد المصفوفة ببيانات افتراضيه للعرض//
  Person[] people = new Person[]
    { new Person("name 1", 4, "23 545 54"),
      new Person("name 2", 3, "14 235 17"),
      new Person("name 3", 7, "16 987 41"),
      new Person("name 4", 18, "n/a"),
      new Person("name 5", 18, "n/a")
    };
وضع المصفوفة في حالة الطلب حيث يستطيع السيرفلت أن يصل لها//
  request.setAttribute("people", people);
%>
```

```
// تنفيذ السيرفلت وإرسال أسم المعامل
// وضع سمك حدود الجدول على القيمة 4
<jsp:include page="/TableServlet" flush="true">
  <jsp:param name="data" value="people"/>
  <jsp:param name="tableOptions"
value="BORDER=4"/>
  <jsp:param name="column" value="name"/>
  <jsp:param name="columnType" value="data"/>
  <jsp:param name="columnHeader" value="Name"/>
  <jsp:param name="column" value="age"/>
  <jsp:param name="columnType" value="data"/>
  <jsp:param name="columnHeader" value="Age"/>
  <jsp:param name="column"
value="getPhoneNumber"/>
  <jsp:param name="columnType" value="data"/>
  <jsp:param name="columnHeader" value="Phone #"/>
</jsp:include>
</body>
</html>
<%!
//يتم تعريف فئة كائن جديد ليحتوى البيانات
  public class Person
  {
    public String name;
    public int age;
    protected String phoneNumber;
    public Person(String aName, int anAge,
      String aPhoneNumber)
    {
      name = aName;
```

```
        age = anAge;
        phoneNumber = aPhoneNumber;
    }
//getPhoneNumber الوصول إلى حقل رقم التليفون عن طريق النداء
 على الوسيلة
بهذه الطريقة نتأكد من سلامة عمل الوسيلة//
    public String getPhoneNumber()
    {
        return phoneNumber;
    }
  }
%>
</body>
</html>
```

استخدام Applet:

تحتوى Applet على الكثير من الأدوات الممكن استخدامها كواجهة للتعامل مع المستخدم من خلال المستعرض وتعطى إمكانيات غير محدودة للتأكد من سلامة البيانات المدخلة من المستخدم أو طريقة عرض البيانات وعادة يتم استخدامهم في جهاز المستخدم وليس جهاز المزود.

وهناك الكثير من المعلومات على الإنترنت التي تستطيع من خلالها معرفة كيفية إنشاء Applets وأنه يجب استخدام الوسم <applet> ولكن لا يوجد في المستعرض إمكانية تنفيذ Applets لذلك حلت شركة SUN هذه المشكلة ضمن مكتبة الجافا JRE حيث يمكنك من خلال الوسم <embd> في المستعرض Netscape والوسم <object> في

المستعرض Internet Explorer من إجبار المستعرض على تشغيل Applet ضمن مكتبة الجافا .

وتقدم لغة JSP إمكانية وضع وسم خاص في الصفحة لاستخدام Applet حيث يقوم الوسم تلقائيا بمعرفة نوع المستعرض ويقوم باستخدام الوسم الصحيح <embd> أو<object> ويقوم بإدخال نصوص HTML الصحيحة وفيما يلي مثال للوسم <jsp: plugin>:

```
<html>
<body>
Here is the applet:
<br>
<jsp:plugin type="applet" code="examples.SwingApplet"
codebase="."
   width="500" height="400">
   <jsp:fallback>
      <p>Unable to use Java Plugin</p>
   </jsp:fallback>
</jsp:plugin>
</body>
</html>
```

النص بداخل الجزء <js:fallback> يتم إظهاره فقط في حالة عدم قدرة المستعرض على تنفيذ Applet إما لمشاكل في تحميل الـــ Applet أو أن المستعرض ليس به إمكانية لذلك ، وإذا أردت تمرير معاملات للـــ Applet يمكنك استخدام <jsp:params> فعلى سبيل المثال:

```
<jsp:params>
   <jsp:param name="param1" value="param1Value"/>
   <jsp:param name="param1" value="param1Value"/>
</jsp:params>
```

الوسم <jsp:params> يجب أن يتم وضعه ضمن الوسم <jsp:plugin> بنفس طريقة الوسم <jsp: fallback>

وفيما يلي نصوص HTML الناتجة عن تشغيل Applet من خلال IE

```
<html>
<body>
Here is the applet:
<br>
<object classid=clsid:8AD9C840-044E-11D1-B3E9-
00805F499D93 width="500" height="400"

codebase="http://java.sun.com/products/plugin/1.2.2/jinsta
ll-1_2_2-win.cab#Version=1,2,2,0">
<param name="java_code"
value="examples.SwingApplet">
<param name="java_codebase" value=".">
<param name="type" value="application/x-java-applet;">
<comment>
<embed type="application/x-java-applet;" width="500"
height="400" pluginspage="http://java
.sun.com/products/plugin/"
java_code="examples.SwingApplet" java_codebase="."/>
<noembed>
    <p>Unable to use Java Plugin</p>
</noembed>
</comment>
</object>
</body>
</html>
```

ويقبل الوسم <jsp:plugin> معظم خصائص الوسم <applet> مثل code, codebase, archive و width و height.

الفصل الثامن

معالجة الأخطاء وإكتشافها

عادة يقابل المبرمج مشاكل في السيرفلت أو صفحات JSP سواء كانت أخطاء أثناء المعالجة أو بعد التنفيذ ، وهذا جزء هام وأساسي في عمل المبرمج فإذا أردت أن تكون مبرمج محترف يجب أن تتعلم حل المشاكل البسيطة والصعبة على السواء.

وإذا قمت أثناء قراءة هذا الكتاب بإجراء أي تجارب على الكود الموجود به فقد تقابل هذه الأخطاء ، وفيما يلي في هذا الفصل سنتعرف على نوع الأخطاء وكيفية القيام بإنشاء ملف تتبع الحركة (log file) وكيف نقوم بمعالجة واكتشاف الأخطاء في السيرفلت أو صفحات JSP ، وأخيرا كيف نقوم بإنشاء صفحة مختصة بك لإظهار معلومات عن الخطأ الحادث.

عملية اكتشاف الأخطاء Debugging:

تكون عمليه Debug عادة صعبة نظرا لأن برامج الجافا تعتمد على استخدام ملفات خارجية كثيرة موجودة على أجهزة مزود عديدة وبالتالي تتنوع الأخطاء ، فقد يكون الخطأ ناتج من نصوص HTML أوكود الجافا أو مشكلة في المزود أو حتى المستعرض نفسه.

وكما رأيت مع JSP فإن برنامج المزود يقوم بعمليات خفية هي عمليه الترجمة والتحويل من JSP إلى سيرفلت فتظهر الأخطاء مع استخدام نصوص الأكواد المختلفة خاصة إذا كان المبرمج لا يفهم طبيعة كائن أو طريقة عمله ، وعادة تكون عملية اكتشاف الأخطاء في صفحة JSP هي عملية لمعالجة الأخطاء في السيرفلت المتولد.

أما عن البرامج التي تعمل بصورة أساسية على أجهزة المزود فعادة تقابل مشكلة مع كثرة الزائرين وهى مشكلة التعامل مع نسخ البرنامج

المحملة في الذاكرة thread أو في مشاكل في التعامل مع ملف قاعدة البيانات من قبل أكثر من جهة فنفس الزائر قد يستخدم نافذتين لتحرير سجل معين في قاعدة البيانات أو زائرين يقوموا بتحرير نفس السجل في نفس المزود.

. وأخيرا قد تحدث أخطاء في البيئة نفسها التي تقوم بتنفيذ البرنامج مما يؤدى لتعطل البرنامج وحدوث أشياء غير متوقعه مثل هذه المشاكل قد تحدث في برنامج المزود نفسه أو كخطأ في ضبط البرنامج على المزود.

★ الأخطاء أثناء التنفيذ وأثناء المعالجة:

تظهر الأخطاء عادة في صفحات JSP سواء أثناء المعالجة أو أثناء التنفيذ ، أما بالنسبة للسيرفلت فتظهر الأخطاء وقت التنفيذ فقط وهذا لأنك تقوم بنفسك بمعالجة السيرفلت ، وفيما سبق درسنا كيف أن JSP تدخل مرحلة ترجمة ثم استدعاء وأثناء الترجمة يقوم برنامج المزود باختبار الكود والتأكد من صحته ثم أنتاج ملف مكتوب بلغة الجافا والملف التنفيذي الخاص به والذي يكون امتداده class. ، وتتم معالجة JSP عند استدعائها أو طلبها أويتم ذلك أثناء عملية النشر أوقد تتم المعالجة أثناء عملية بناء الحزمة باستخدام أداة مثل Ant وأداة jspc .

وفيما يلي أكثر المشاكل التي قد تحدث لك أثناء الترجمة ، فالشكل التالي يظهر خطأ ناتج عن كتابة كلمة المتجهة page بحروف خطأ.

```
HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

org.apache.jasper.JasperException: /debuggingpractice.jsp(1,5) Invalid directive
	org.apache.jasper.compiler.DefaultErrorHandler.jspError(DefaultErrorHandler.java:83)
	org.apache.jasper.compiler.ErrorDispatcher.dispatch(ErrorDispatcher.java:402)
	org.apache.jasper.compiler.ErrorDispatcher.jspError(ErrorDispatcher.java:126)
	org.apache.jasper.compiler.Parser.parseDirective(Parser.java:548)
	org.apache.jasper.compiler.Parser.parseElements(Parser.java:1625)
	org.apache.jasper.compiler.Parser.parse(Parser.java:173)
	org.apache.jasper.compiler.ParserController.parse(ParserController.java:247)
	org.apache.jasper.compiler.ParserController.parse(ParserController.java:149)
	org.apache.jasper.compiler.ParserController.parse(ParserController.java:135)
	org.apache.jasper.compiler.Compiler.generateJava(Compiler.java:243)
	org.apache.jasper.compiler.Compiler.compile(Compiler.java:451)
```

تظهر رسالة الخطأ بصورة واضحة سبب الخطأ "Invalid directive" وقام المعالج بتقديم رقم السطر والعمود في مكان خطأ بين أقواس .

ولتوضيح نوع آخر من الأخطاء التي تحدث أثناء توليد السيرفلت نرى الشكل التالي الذي يظهر خطأ ناتج عن محاولة تنفيذ وسيلة من كائن غير موجود.

```
HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

org.apache.jasper.JasperException: Unable to compile class for JSP

An error occurred at line: 5 in the jsp file: /debuggingpractice.jsp

Generated servlet error:
    [javac] Compiling 1 source file

/opt/jakarta-tomcat-5.0.12/work/Catalina/localhost/_/org/apache/jsp/debuggingpractice_jsp.java:45: cannot
symbol   : variable otu
location: class org.apache.jsp.debuggingpractice_jsp
        otu.println("Hello Debugging Practice");
        ^
1 error
```

في هذه الحالة أيضا نجد أن برنامج المزود Tomcat يظهر بوضوح صفحة الخطأ ويحدد نوع الخطأ ومكانه .

ولكن هذا ليس حال معالجة الأخطاء أثناء التنفيذ فالأمر يتطلب بعض العمل اليدوي ، وبالنسبة للصفحات JSP والسيرفلت فإن معالجة أخطاء التنفيذ للاثنين متماثلة ، فأخطاء التنفيذ تنتج عادة عن أخطاء في منطق البرمجة وأحيانا يكون اكتشاف الخطأ سهل إذا نتج عنه رسالة خطأ تسمى exception ولكن في أحيان أخرى سترى مجرد صفحة بيضاء أو يقوم البرنامج بتصرفات غير متوقعة لذلك سنقوم في القسم التالي بتوضيح كيفية معالجة هذا النوع الأخطاء.

★ استخدام ملف التتبع Log File:

طريقة معالجة الأخطاء من الطرق القديمة ولكنها لا تزال مستخدمة ، ورغم وجود برنامج مكتشف الأخطاء (debugger) الآن إلا أن البرنامج قد يعمل بصورة جيده مع debugger ثم يتوقف عندما يعمل بدونه لذلك يتم استخدام ملف التتبع الذي يمكننا من معرفة القيم التي يأخذها المتغير طوال سير البرنامج.

وهذا يعطينا صورة أوضح عن طريقة عمل البرنامج بدون التدخل في أثناء تنفيذه لعمليه معينه وإيقافه باستخدام debugger ولكن يجب عدم كتابة الكثير من المعلومات في الملف لأن هذا قد يسبب بطئ التنفيذ.

★ العبارتين system.out, system.err:

بالنسبة لبرامج الجافا التنفيذية التي تعمل بصوره منفصلة يمكنك تتبع الخطأ عن طريق العبارتين system.out, system.err وهذا يحدث عندما

تقوم باستخدام العبارة printStackTrace من الكائن Throwable فيتم استخدام العبارة system.err أو غيرها حسب احتياجك .

أما بالنسبة لصفحات JSP والسيرفلت فإن رسالة الخطأ قد لا تظهر بالضرورة على الشاشة ويعتمد الأمر على محرك JSP المستخدم ، فبالنسبة للمزود Tomcat يتم كتابة الرسالة في الدليل logs أسفل الدليل الرئيسي للـــTomcat ولا يدعم كل المزودات استخدام العبارتين system.out, system.err.

لاحظ أنه إذا كان السيرفلت منحدر من الكائن GenericServlet أو الكائن HttpServlet فيمكنك استخدام وسائل الكائن log من السيرفلت نفسها ويمكنك أيضا استخدام الكائن ServletContext من خلال المتغير الضمني application.

ووسائل الكائن log هي:

```
public void log(String message)
public void log(String message, Throwable throwable)
```

لاحظ أن التعريف الثاني للوسيلة log يمكنك من طباعة رسالة وأيضا محتويات الذاكرة التي حدثت بها الخطأ.

✶ استخدام الحزمة Log4J:

تستخدم الحزمة Log4J كأداة أكثر تفاعليه مع عملية التتبع فهي تعطى إمكانية لتعريف تقسيم معين للمعلومات الواردة بالملف وتحديد مكان الملف وتشكيل الرسالة التي سوف تسجل بالملف ويمكنك معرفة المزيد عن هذه الحزمة من الرابط التالي

http://jakarta.apache.org/log4j/docs/index.html.

وتقوم الحزمة Log4J بتعريف ثلاث مكونات رئيسيه:

Loggers و Appenders ويتم استخدامهم لإخراج الرسائل لمكان محدد

Levels: عندما تقوم بتعريف الرسائل تقوم بتحديد أسم والمستوى الخاص بها وفيما يلي المستويات التي يمكن استخدامها:

DEBUG: وهو المستوى الرئيسي يستخدم لتحديد معلومات تفصيلية عن سير البرنامج

INFO: يستخدم لتقديم معلومات عن طريقة عمل البرنامج

WARN: يستخدم لتحديد إمكانية حدوث تأثيرات خطأ في سير البرنامج

ERROR: لتحديد إمكانية وجود أخطاء في البرنامج

FATAL: يدل على أن البرنامج به خطأ لا يمكن إصلاحه وقد يؤدى إلى إنهاء البرنامج

ALL: ليدل على إظهار كل الرسائل

OFF: لإبطال إظهار الرسائل

ويستخدم المكون loggers المستويات لتحديد إرسال الرسالة أم لا فمثلا إذا تم إنشاء logger مع المستوى DEBUG فسوف ينتج عن هذا إرسال معلومات لكل مستوى ولكن إذا تم إلحاقه مع المستوى ERROR فسوف تقوم فقط بتسجيل الرسائل من نوع ERROR أو FATAL فقط , وتحتوى الفئة Logger على وسائل تتعلق بكل مستوى ولكل منها شكل مثل التالي :

```
void debug(Object message)
void debug(Object message, Throwable t)
```

الوسائل السابقة تأخذ كائن يتم تحويله إلى نص حرفي واختيار المعامل Throwable الذي يقدم معلومات إضافية عن الخطأ ، بالأضافه

للوسائل debug, info, warn, error, fatal فإن الفئة Logger لديها الوسيلة log والتي يمكنها إنشاء رسائل تتبع بمختلف المستويات .

أما المكون Appenders فيقوم بتحديد ما إذا كانت الرسالة يجب أن يتم إرسالها بناء على المستوى المحدد للرسالة ومكان ملف التتبع وتقدم الحزمة Log4J إمكانية للتتبع داخل نافذة أو في ملف أو في قاعدة بيانات ويمكنها أيضا إرسال المعلومات إلى ايميل معين أو جهاز خادم بمكان آخر.

ويمكنك ضبط الحزمة Log4J برمجيا وتعريف مكونات جديدة بها من خلال ملف الخصائص properties وفيما يلي مثال للملف :log4j.properties

```
log4j.category.examples.logger=DEBUG, file
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=<TOMCAT_HOME>/webapps/lo
gs/log.txt
log4j.appender.file.layout=org.apache.log4j.SimpleLayout
```

لقد قمنا في السطر الأول بتعريف logger اسمه examples.logger والذي يقوم بتسجيل الأحداث على المستوى DEBUG أو أعلى والمكون appender اسمه file . السطر الثاني والثالث يتم تعريف appender بهما من النوع RollingFileAppender والمخرجات ستكون في المسار /webapps/logs المتفرع من الدليل الرئيسي للـــ Tomcat ولا تنسى أن تقوم باستبدال العبارة <TOMCAT_HOME> بالدليل الفعلي الذي يوجد به Tomcat وسيتم بناء على الضبط RollingFileAppender نسخ احتياطي لملف التتبع عندما يصل حجمه إلى الحجم المحدد . والسطر الأخير يقوم بتحديد شكل المخرجات على الهيئة SimpleLayout.

ولكي تستطيع استخدام Log4J مع Tomcat فيجب أن تنسخ الملف log4j.jar إلى الدليل /common/lib/ في الدليل الذي تم فيه تركيب Tomcat

وفيما يلي مثال لسيرفلت يقوم باستخدام الحزمة Log4J لتسجيل رسائل التتبع:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import org.apache.log4j.*;
public class Log4JServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
HttpServletResponse response)
     throws ServletException, IOException {
    Logger logger =
Logger.getLogger("examples.logger");
    logger.info("In the doGet method of Log4JServlet!");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>A Servlet That uses
Log4J</title></head>");
    out.println("<body>");
    out.println("<p>We've just logged an event!</p>");
    out.println("</body></html>");
  }
}
```

الملف log4j.properties الخاص بالبرنامج يجب وضعه في الدليل WEB-INF/classes من الدليل الفرعي للبرنامج وعن طريق ذلك يمكنك ضبط كل برنامج بصوره مستقلا عن الآخر.

لاحظ أنه بدءا من الإصداره Java 1.4 يوجد إمكانيه مدمجه للتتبع وهى تشبه كثيرا طريقة التعامل مع الحزمة Log4J ، ويمكنك أيضا استخدام ملف خارجي لتسجيل رسائل التتبع ويجب استخدام خاصية الفتح append حتى تقوم بإضافة رسائل للملف بدلا من إلغاء الرسائل القديمة في كل مرة تقوم فيها باستخدام الملف .

★ التعقب باستخدام الأخطاء Exceptions :

أحيانا عندما تقوم بعمليه معالجة الأخطاء تريد أن تكتشف أي إجراء أو كود قام باستدعاء وسيله معينه ، فيمكنك في هذه الحالة أن تقوم بإطلاق رسالة خطأ exception ولكنك لا تريد أن يتوقف البرنامج أثناء التنفيذ ولكن أن يتم تسجيل رسالة التتبع لتدل على الإجراء الذي قام بالنداء على الوسيلة فمثلا إذا كان هناك كائن قام بالنداء على وسيله وجعلها تأخذ قيمة خطأ وتريد معرفة الكائن المسئول عن هذا .

ويمكنك إطلاق خطأ أثناء التنفيذ وتعقبه في الحال مما يمكنك من أن تأخذ صورة واضحة عن الذاكرة stack ويتم لك كما يلي:

```
try
{
  throw new Exception("any error");
}
catch (Exception exc)
{
```

```
System.out.println("The Class which called this method is:");
exc.printStackTrace(System.out);
}
```

هذه التقنية ليست الأفضل ولكن أحيانا لن تجد غيرها لتصل لمكان الخطأ.

★ **استخدام Debugger:**

تظهر فائدة الـ Debugger عندما تريد أن تختبر حالة البرنامج أثناء تنفيذه فيعطيك الـ Debugger إمكانية لمعرفه حالة كائن معين أو تتبع التغييرات أو معرفة أي الأحداث تم تنفيذها .

ويوجد العديد من Debugger للغة الجافا ، فحزمة JDK تأتى مع أداة لكتابة سطور الأوامر تسمى jdb ولا تحتوى هذه الأداة على واجهة للتعامل مع المستخدم ، وتظهر فائدة هذه الأداة عندما تريد اختبار البرنامج في أجهزة خادم موجودة بمكان آخر ولا تستطيع أن تصل إليه أو استخدام أي برنامج مرئي للتعامل معها بسبب وجود حماية على أجهزة الخادم Firewall لذلك قد تستخدم أداة الاتصال telnet ومن خلالها تستطيع استخدام الأداة jdb.

لاحظ أنه إذا لم تستطع استخدام نظام X Window الخاص بالتعامل مع أجهزة الخادم بسبب وجود Firewall فيمكنك استخدام الأداة (Virtual Network Computing) من الرابط http://www.realvnc.com حيث يمكنه الاتصال بأجهزة الخادم من خلال Firewall والأجهزة التي تعمل على نظام التشغيل Windows تماما مثل الأداة الشهيرة Norton PCAnywhere .

وعموما هناك طريقتين أساسيتين لاختبار البرنامج من خلال الـ Debugger الأولى هي تشغيل البرنامج من خلال الـ Debugger والثانية تشغيل البرنامج مع ضبط إمكانية الاتصال الشبكي remote debugging ثم تقوم بإلحاق debugger بالبرنامج ، والميزة في اختبار البرنامج عن طريق الاتصال الشبكي هي إطلاق الحرية لعمل جهاز الخادم في تشغيل البرنامج ثم نقوم بالاتصال بالبرنامج عن طريق برنامج debugger ولكن هذه الطريقة معقده وعرضه لظهور الأخطاء.

★ **طريقة اختبار السيرفلت باستخدام jdb:**

تعطى الأداة jdb الإمكانية لتنفيذ سطر الكود مع إظهار قيم المتغيرات ولاستخدام هذه الأداة مع برامج الجافا قم بكتابة اسم الأداة ثم اسم البرنامج كما يلي:

```
jdb myServlet
```

حيث myServlet هي السيرفلت المراد اختبارها ، ويمكنك بنفس الطريقة اختبار السيرفلت التي تعمل على المزود Tomcat فيجب علينا النداء على الملف Catalina.bat الذي يحتوى على أوامر تشغيل وبدء Tomcat بالنسبة للنظام Windows أما للنظام يونيكس أو لينيكس فاستخدام الملف Catalina.sh وقم بكتابة الأمر كما يلي:

```
catalina debug
```

وبالنسبة لليونيكس:

```
catalina.sh debug
```

ولكي تقوم باختبار سيرفلت فكل ما عليك هو أن تخبر jdb أن يوقف البرنامج عند تنفيذ الوسيلة Get do كما يلي:

```
stop in myServlet.doGet
```

ثم قم بعد ذلك بتشغيل Tomcat بالأمر التالي:

```
run
```

وعندما تقوم باستدعاء السيرفلت يقوم الـــ debugger بإظهار رسالة بأنه وصل للعملية التي تم وضع علامة التوقف breakpoint بها ، وتستطيع أن تظهر السطر الحالي في الكود عن طريق الأمر list ولكن يجب أولا أن تحدد مسار الملف المصدر ويتم ذلك كما يلي:

```
use /home/myServlets/
```

ثم بعد ذلك أكتب الأمر list سيتم عرض قائمة بها بعض سطور الكود ويظهر سهم => بجانب السطر الذي تم التوقف عنده.

وتحتوى الأداة jdb على الكثير من الأوامر التي يمكن التعرف عليها عن طريق كتابة الأمر help أو العلامة ? وحتى يتم استكمال تنفيذ البرنامج يجب كتابة الأمر cont ثم الأمر quit للخروج.

وعن طريق الاتصال بجهاز الخادم يمكنك استخدام الأداة jdb لاختبار البرنامج به وإذا كان Tomcat موجود على جهاز آخر غير الذي تعمل عليه فيمكنك تشغيل الـــTomcat في الجهاز الخادم عن طريق الأمر التالي:

```
catalina.sh jpda start
```

وهذا يجعل المزود Tomcat يعمل من خلال حزمة JVM التي تدعم الاختبار عن طريق الحزمة JPDA وهى اختصار Java Platform Debugging Architecture وهذا ما يجعل إمكانية الاختبار لجهاز متصل بأخر ممكنه ، ثم قم بكتابة الأمر التالي من الجهاز الذي تعمل عليه

```
jdb -connect
com.sun.jdi.SocketAttach:hostname=HOSTNAME,port=8
000
```

حيث العبارة HOSTNAME تستبدل باسم الخادم الذي يعمل به Tomcat وبعد ذلك يكون اختبار البرنامج يماثل تماما اختبار البرنامج على جهازك ولكن بدون استخدام الأمر run لأن Tomcat تم تشغيله بالفعل.

وكقاعدة أي برنامج معالجة الأخطاء debugger يدعم الحزمة JPDA سيكون قادر على معالجة السيرفلت التي تعمل داخل Tomcat.

★ استخدام معالج الأخطاء netBeans IDE 4.0:

يعتبر البرنامج Beans net من أفضل وأقوى البرامج التي تتعامل مع JSP والسيرفلت حيث يمكنك استخدامه لتطوير سيرفلت ومعالجة الأخطاء ويأتي مجانا مع حزمة JDK الكاملة (50 MB) ويمكن الوصول إليه وتحميله عن طريق الموقع الرسمي له http://www.netbeans.org.

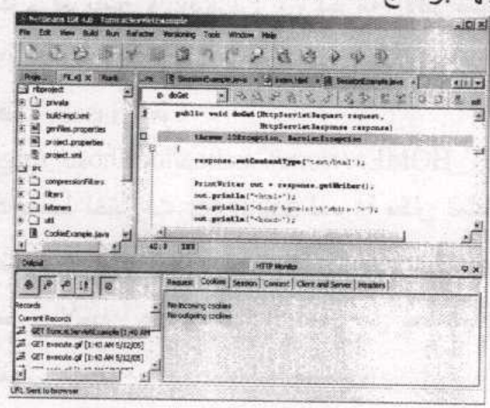
وهناك عدة برامج أخرى جيده نذكر منها :

- Communiqué JSP Debugger وهو برنامج مجاني ويمكنه معالجة الأخطاء في الملف المصدر ويمكن الوصول إليه عن طريق الرابط http://www.day.com/content/en/product/productline/unify/ide/dnlogin.html
- Eclipse وهو برنامج آخر مجاني ويعتبر من البرامج الرائعة مفتوحة المصدر ، ويمكن استخدامه مع JSP والسيرفلت إذا قمت بإضافة البرنامج إليه، ويمكنك معرفة المزيد عنه من الرابط http://www.eclipse.org
- Swat J برنامج لمعالجة الأخطاء مفتوح المصدر .ويتميز بالسهولة الشديدة ويمكن الوصول إليه عن طريق الرابط:

. http://www.bluemarsh.com/java/jswat/

وبالرغم من أن برامج معالجة الأخطاء debuggers تختلف في إمكانياتها وطريقة عمل كل منها إلا أنها تتشابه في جوانب كثيرة بالنسبة للتعامل مع JSP والسيرفلت ، فعادة تبدأ معالجة الأخطاء عن طريق إنشاء مشروع Project أو جلسه Session حيث يتم تعريف مكان الملفات المصدر والملفات التنفيذية Classes ومعظم هذه البرامج يمكنهم التضامن مع برنامج Tomcat بطريقة سهله عندما تتم المعالجة على نفس الجهاز ، فلديهم القدرة على بدء وإيقاف Tomcat وأحيانا نشر البرنامج كله على الإنترنت . وجميع البرامج متوافقة مع الحزمة JPDA ومثل الأداة jdb فكل ما تحتاجه هو أن تقوم بتشغيل Tomcat مع وضع تمكين (enabled) للحزمة JPDA وتقوم بكتابة أسم وIP للجهاز الذي تقوم بتشغيل Tomcat به ، وبالنسبة للملفات المصدر كل ما عليك هو تقديم مسار لهذه الملفات حتى يتم ظهور السطر الحالي الذي تم الوقف عنده كما رأيت مع jdb.

وفيما يلي واجهة برنامج Beans net



وكما ترى إذا كنت مستخدم لأحد لغات البرمجة المرئية يمكنك أن تتعرف على نوافذ البرنامج بسهوله فيمكنك وضع علامات التوقف Breakpoints ومعرفة السطر الحالي للتنفيذ ورؤية ومتابعة التغير في قيمة متغيرات معينه كل هذا بقوة وسهوله لذلك يفضل استخدام برامج المعالجة المرئية عن سطر الأوامر.

معالجة أخطاء صفحات JSP:

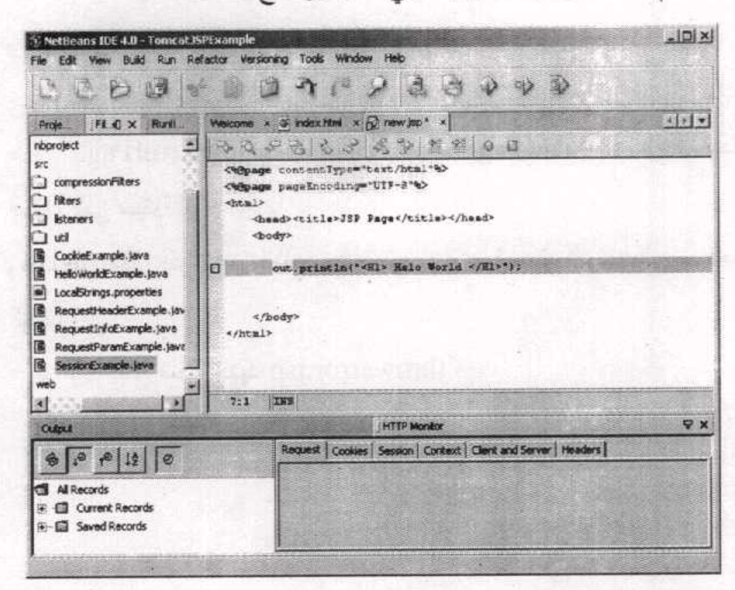
لأن صفحات JSP تصبح بعد ذلك سيرفلت فيمكنك معالجة أخطاء صفحة JSP بنفس الطريقة مع السيرفلت مع بعض الخطوات الإضافية.

عندما يقوم Tomcat بترجمة صفحات JSP فإنه يقوم بوضع السيرفلت الناتج في الدليل الرئيسي للـــ Tomcat وهو المكان الذي يجب أن تبحث فيه عن السيرفلت ، فإذا نظرت إلى هذا الدليل فيمكنك أن ترى الدليل /Catalina/localhost أسفل الدليل /work ولاحظ أن هناك أدلة لكل نسخة برنامج حيث النسخة الأساسية تبدأ بالحرف (_) . وبعد الدليل الذي يوجد به نسخة البرنامج الذي تقوم بمعالجته ستجد نفس النظام الهرمي مستخدم مع حزم البرنامج Packages . والحزمة الافتراضية لصفحات JSP هي org.apache.jsp لذلك يكون المسار الكامل لصفحات JSP وملفات السيرفلت Classes هو:

<TOMACT_HOME>/work/Catalina/localhost/_/org/apache/jsp

بعد ذلك يمكنك ضبط اختيارات debugger لمعرفة مكان الملفات المصدر وفى حالة استخدام Tomcat فإن Tomcat قد يحدد مكان هذه الملفات تلقائيا وفى بعض الأحيان عند معالجة الأخطاء عن بُعد يتم إنشاء نسخه محليه من الملف.

ومن ضمن إمكانيات Net Beans إمكانية معالجة الأخطاء باستخدام ملفات JSP فقط وفيما يلي شكل يوضح ذلك:



وسائل التعامل مع الأخطاء:

أحيانا لن تجد الخطأ الذي قد يسبب مشكلة أثناء سير البرنامج مهما حاولت مع معالج الأخطاء ومن ضمن الوسائل الفعالة لمعالجة ذلك هو استخدام العبارة try-catch التي تتحكم في خط سير البرنامج إذا حدث خطأ معين بحيث يمكن توجيه سير البرنامج إلى صفحة خاصة لمعالجة الخطأ أو إظهار معلومات عنه .

ولا يجب أن تقوم بكتابة عبارة try-catch في كل مرة تريد فيها أن تمنع ظهور خطأ معين من إيقاف البرنامج ولكن يمكنك تحديد صفحة للخطأ في كل صفحة JSP وعندما يتم حدوث الخطأ يقوم محرك JSP تلقائيا بالتحول

إلى هذه الصفحة وبعد ذلك يمكن عن طريق صفحة الخطأ تسجيل الخطأ الحادث والذي يتم تمريره عن طريق المتغير الضمني exception وإظهار ستجابه مناسبة للمستخدم .

ولكي تقوم بتنفيذ ذلك قم باستخدام عبارة التوجيه <%@ page errorpage="myErrorPage" %> حيث myErrorPage هي صفحة الخطأ التي سيتم التوجه إليها .

وفيما يلي مثال نقوم فيه بإنتاج خطأ حتى نرى كيف يتم التوجه إلى صفحة الخطأ

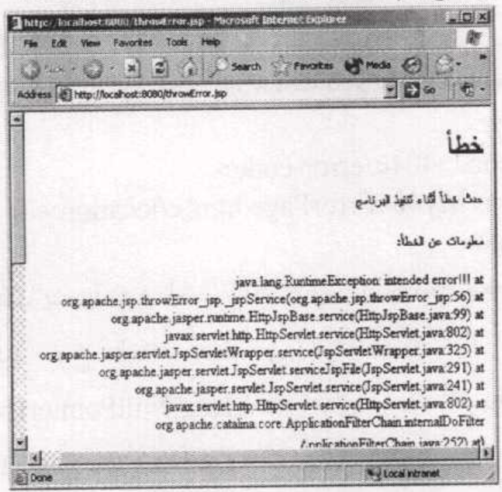
- قم أولا بفتح ملف جديد throwerror.jsp:

```
<%@ page errorPage="errPage.jsp" %>
<html>
<body>
<h1> Hello World!</h1>
</body>
</html>
<%
//إنتاج خطأ لاختبار صفحة الخطأ
// نقوم بكتابة أي تعبير لمنع الخطأ
// "Statement Not Reached"
  int x = 1;
  if (x == 1)
  {
     throw new RuntimeException("intended error!!!");
  }
%>
```

- قم بإنشاء صفحة الخطأ errPage.jsp:

```
<%@ page isErrorPage="true" contentType="text/html; charset=windows-1256" language="java" import="java.io.*" %>
<html dir="rtl">
<body bgcolor="#ffffff">
<h1>خطأ</h1>
حدث خطأ أثناء تنفيذ البرنامج
<p>
معلومات عن الخطأ:
</p>
<% exception.printStackTrace(new PrintWriter(out)); %>
</body>
</html>
```

وعند تنفيذ البرنامج يتم التحول إلى صفحة الخطأ كما بالشكل



لاحظ أنه إذا كانت الصفحة التي تم حدوث الخطأ بها تحتوى على عناصر أخرى مثل صور أو نصوص فإن هذه العناصر قد تظهر قبل الاتجاه

لصفحة الخطأ لذلك يفضل دائما كتابة الكود الذي قد ينتج عنه خطأ في أول الصفحة.

بعض الأخطاء لا يمكن الكشف عنها سواء في صفحة JSP أو في سيرفلت فمثلا إذا كنت تستخدم نوع جديد من الوسوم في ملف HTML فإن خادم الويب يقوم بإنتاج رسالة خطأ "HTTP 404 error" ولأن ليس هناك سيرفلت أوJSP فإن معرفتك بالخطأ قد تأتى عن طريق إخبار المستخدم لك بذلك .

ولكن لحسن الحظ فإن دوال API الخاصة بالسيرفلت تعطى لك إمكانية لتصيد أخطاء خادم الويب أو أخطاء الجافا ويجب أن يتم وضع هذه الكود في ملف يسمى deployment descriptor وهو ملف يحتوى على نصوص لتحديد محتويات برنامج الويب ويمكنك الرجوع إلى ملفات المساعدة في كيفية إنشاء هذه الملفات.

ولتقوم بكتابة كود لمعالجة الخطأ 404 فأنك تقوم بكتابة المعلومات التالية

```
<error-page>
   <error-code>404</error-code>
   <location>My404ErrorPage.html</location>
</error-page>
```

لاحظ أننا قمنا بتحديد كود الخطأ 404 الذي نريد التعامل معه ولكن يمكنك أيضا تحديد نوع الخطأ بدلا من الرقم فمثلا إذا حدث الخطأ java.lang.NullPointerException الذي يحدث نتيجة عدم وجود كائن فيمكنك تصيد هذا الخطأ بكتابة النص التالي:

```
<error-page>
   <exception-
type>java.lang.NullPointerException</exception-type>
```

```
  <location>NullPointer_ErrorPage.jsp</location>
</error-page>
```

وصفحة الخطأ هنا قد تكون ملف HTML أو صفحة JSP أوسيرفلت أو أي شيء آخر يمكن عرضه.

ومن هنا نرى أن لغة الـــ JSP والجافا متشعبة وليس لإمكانياتها نهاية .

الفصل التاسع

استخدام مكتبة وسوم الجافا
JAVA TAG LIBRARY

الوسوم المدمجة في المكتبات من أهم الأدوات التي يمكن استخدامها لإنشاء وتحرير ملفات JSP بصوره تلقائية.

وتعتبر عملية فهم وتعديل صفحه JSP التي تحتوى على كود جافا من الأشياء العسيرة وتأتى هنا مكتبة وسوم الجافا الجاهزة التي تقلل كثيرا الحاجة لاستخدام كود الجافا عن طريق وسوم تشبه كثيرا وسوم لغة HTML وتؤدى معظم الوظائف القياسية وباستخدامها يمكنك كتابة برامج JSP بدون الحاجة لكتابة أي كود جافا.

وتستخدم لغة JSP أدوات قياسية تسمى standard actions مثل <jsp:getProperty> و<jsp:useBean> وهى تشبه لغة HTML وتستخدم عادة لإنشاء أدوات ديناميكية أو النداء على صفحات أخرى ودائما يكون هناك حاجة لأدوات أخرى . لذلك يمكنك كمبرمج جافا أن تقوم بزيادة هذه الأدوات مبتدءا بكائنات رئيسيه Classes وتسمى هذه الأدوات custom action والذي يمكنه أن يقوم بتنفيذ أي شيء مثل الوصول إلى معلومات الطلب وتعديل بيانات الاستجابة كما يمكنها الوصول بحريه إلى دوال API واستخدام الموارد الخارجية مثل قواعد البيانات وخوادم الايميل أوLDAP وتصميم custom action يجعل برمجة صفحات JSP سهله وسريعة .

ويتم دمج custom action في الصفحة باستخدام نصوص تشبه HTML وهى نصوص عناصر XML وفى الواقع فإن custom action هي من كائنات الجافا Java Classes والملف الذي سوف تحتاجه لاستدعاء هذه الكائنات موجود بملف اسمه Tag Library Descriptor أوTLD ومكتبة custom tag library هي في الواقع مؤلفه من ملفات

TLD والكائنات الأخرى Classes مضغوطة في ملفات JAR ليسهل تركيبها على الخادم كخلفيه مبدئية عن custom action ، ويمكنك أن تعرف كيف يتم الاستجابة لهذه الأدوات حيث يوجد في الجافا كائن يسمى tag handler الذي يحتوى على كود custom action ويقوم tag handler بالانحدار من الواجهة javax.servlet.jsp.tagext.Tag مباشرة أو عن طريق كائن مدعم .

مثال:

```
package com.ora.jsp.tags.motd;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import com.ora.jsp.beans.motd.*;
public class MixedMessageTag extends TagSupport {
private MixedMessageBean mmb =
new MixedMessageBean( );
// Attributes
private String category;
public void setCategory(String category) {
this.category = category;
}
public int doEndTag( ) {
mmb.setCategory(category);
JspWriter out = pageContext.getOut( ).
try {
out.println(mmb.getMessage( ));
}
catch (IOException e) { }
return EVAL_PAGE;
}
```

}

في المثال السابق لكل خاصية يقوم tag handler بتعريف وسيلة (method) تستطيع أن تقوم بتغيير قيمة الخاصية مثل الوسيلة setCategory() ويقوم خادم الويب باستدعاء الوسائل المعرفة في الواجهة Tag مثل الوسيلة doEndTag() ويقوم tag handler بتنفيذ الكود الخاص بالوسيلة.

والسؤال الآن لماذا تسمى custom tag library وليس مجموعة أداوت custom actions ؟ وسبب هذا هو التسمية الطويلة custom action element التي تعبر عن عناصر هي في الواقع tags لذلك لتسهيل النطق والكتابة تم استخدام التسمية custom tag ولكن أي من التسميتين لهم نفس المعنى.

كيف نقوم بتجهيز JSTL:

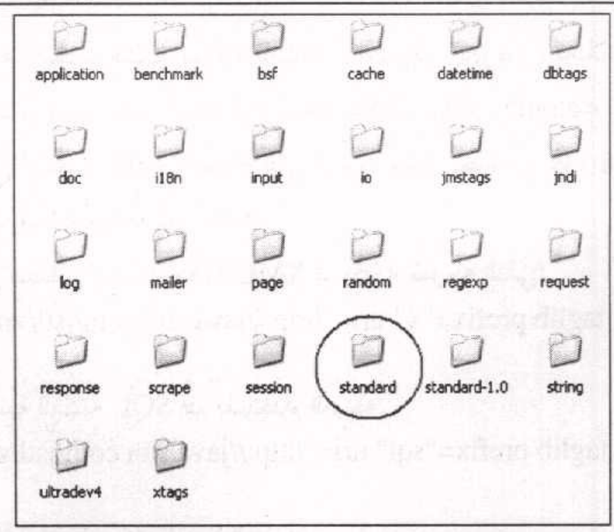
أسهل الطرق لأداء ذلك هو تحميل الملف jakarta-taglibs-20050529.zip من الموقع التالي:

http://jakarta.apache.org/taglibs/index.html

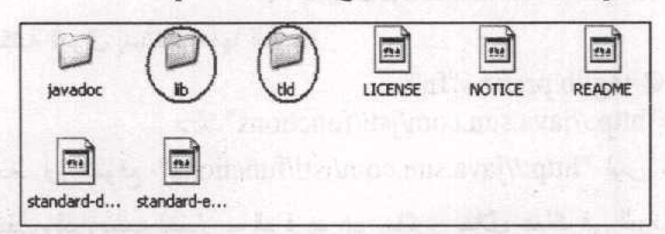
ويمكنك تحميل نسخه مشابه من الموقع :

http://java.sun.com/products/jsp/jstl

لاحظ انه في الوقت الذي ستقوم فيه بتحميل الملف سيتغير أسم الملف تبعاً لتاريخ اليوم (20050529) بعد ذلك قم بفك الملف بأحد البرامج التي تتعامل مع الامتداد zip ويجب أن ترى هيكل الادله كما بالشكل:



قم بفتح الدليل standard والذي سوف يحتوى على ملفات JAR وملفات TLD التي سوف نحتاجها كما يتضح من الشكل التالي:



ويجب وضع محتويات هذا الدليل (copy and paste) في الدليل -WEB INF/ الخاص بأي برنامج تعمل به وبالنسبة لنا الآن سنقوم بوضعه في الدليل ROOT\WEB-INF .

ولاستخدام المكتبة Core قم باستخدام الموجه التالي:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core"
%>
```

نقوم هنا بتحديد الكلمة المبدئية للمكتبة core وهي الحرف c فمثلا إذا قمنا باستخدام وسم يسمى out فإن اسمه بالكامل يكون <c:out> ويمكنك بالطبع استخدام الكلمة المبدئية التي تريدها ولكن يجب أن تقوم باستخدام كلمات مبدئية مختلفة لكل مكتبه.

ولكي تستطيع استخدام مكتبة XML قم بكتابة الموجه التالي:

```
<%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml"
%>
```

وبالنسبة للمكتبة SQL فقم باستخدام الموجه

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql"
%>
```

والمكتبة Fmt يستخدم معها الموجه

```
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jstl/fmt" %>
```

ومكتبة الدوال يستخدم معها الموجه

```
<%@ taglib prefix="fn"
uri="http://java.sun.com/jstl/functions" %>
```

لاحظ أن الموقع "http://java.sun.com/jstl/functions" ليس موقع حقيقي ولكن يجب اختيار موقع فريد غير متكرر خلال عملك في البرنامج ويجب أيضا تضمين المكتبة بالملف web.xml فمثلا إذا كنت تريد تضمين المكتبة core يتم ذلك عن طريق الكود التالي

```
<taglib>
  <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
  <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

مكتبه الوسوم Core:

هذه المكتبة تتضمن وسوم لقراءة وتغيير وعرض قيم الخصائص وتنفيذ وسوم أخرى حسب شروط معينه وفيما يلي أهم أنواع هذه الوسوم:

– وسوم لجميع الإغراض:

تتيح لك هذه الوسوم إضافة أو حذف متغيرات أوعرض قيم هذه المتغيرات وأيضا إمكانيه وضع مجموعه من الوسوم في قالب try-catch

الوسم <c:out>:

يقوم هذا الوسم بإظهار نتيجة تعبير معين مثل الوسمين <%= %> ولكن مع ثلاثة اختلافات الوسم <c:out> يتيح لك استخدام النقطه "." للوصول إلى الخصائص فمثلا للوصول إلى الخاصية customer.address.street فإن التعبير باستخدام الوسمين <%= %> يكون:

```
<%= customer.getAddress().getStreet() %>
```

وباستخدام الوسم <c:out> يكون:

```
<c:out value="customer.address.street"/>
```

والاختلاف الثاني هو أن الوسم <c:out> يمكنه أن يتجاهل وسوم XML بدون أن يحدث تعارض ويعتبره وسوم حقيقية ، فمثلا إذا كنت تريد أن تظهر النص الحرفي <Hello> في ملف XML فيجب أن تقوم باستخدام الحروف

```
&lt;hello&gt;
```

أو الحروف

```
<[CDATA[<Hello>]]>
```

أما الوسم <c:out> فيتضمن خاصية تسمى escapeXML فإذا كانت قيمة هذه الخاصية True فيتم تلقائيا تجاهل النص لعدم تقييمه كنص XML كما يلي:

```
<c:out                    value="customer.address.street"
escapeXML="true"/>
```

والاختلاف الأخير هو أن الوسم <c:out> يتيح لك وضع قيمه افتراضية يتم استخدامها وعرضها إذا كانت نتيجة التعبير تقوم بإظهار القيمة Null أما الوسم <% =%> فيقوم بإظهار القيمة Null كنص حرفي null أما الوسم <c:out> فيمكنك من تحديد قيمه للعرض في حالة القيمة Null فيمكنك أن تقوم بتحديدها N/A أو None أو حتى نص خالي ويمكنك تنفيذ ذلك بطريقتين كما يلي:

```
<c:out value="customer.address.street" default="N/A"/>
```

أو تحديد القيمة الافتراضية داخل الوسم <c:out> نفسه كما يلي:

```
<c:out value="customer.address.street">
 No address available
</c:out>
```

ويمكن للوسم <c:out> إظهار قيمة افتراضية كنتيجة لعمليات قامت بها وسوم أخرى فمثلا الوسم <fmt:formatDate> يتيح لك تخزين التاريخ بتشكيل معين في متغير فإذا كان التاريخ null فإن قيمة المتغير تكون null وبالرغم من أن الوسم <fmt:formatDate> لا يملك خاصية افتراضية فيمكنك استخدام الوسم <c:out> لإظهار التاريخ بتشكيل معين مع قيمة افتراضية كما يلي:

```
<fmt:formatDate var="dateVar" value="${dueDate}/>
```

```
<c:out value="${dateVar}" default="No due date specified"/>
```

✶ الوسم <c:set>:

يقوم هذا الوسم بإنشاء متغيرات جديدة تحتوى على قيمة كنتيجة لتعبير معين ويوجد صيغتين مختلفتين للوسم :

الصيغة الأولى حيث يمكنك تحديد قيمة للمتغير كما يلي:

```
<c:set var="variableName" value="expression"/>
```

حيث variableName هو اسم المتغير والقيمة expression هي التعبير الذي سيتم تخزين قيمته في المتغير .

مثال:

```
<c:set var="custAddr" value="${customer.address}"/>
```

ويمكنك أيضا وضع التعبير والمتغير داخل جسم الوسم <c:set> وميزة هذه الطريقة أنه يمكنك استخدام وسم آخر داخل الوسم <c:set> مثل الوسم <c:out> كما يلي:

```
<c:set var="custAddr>
  <c:out value="${customer.address.street}"/><p>
  <c:out value="${customer.address.city}"/>
  <c:out value="${customer.address.state}"/>,
  <c:out value="${customer.address.zip}"/>
</c:set>
```

وافتراضيا فإن المتغير الذي يقوم الوسم <c:set> بإنشائه له مدى الصفحة الحالية ولكن يمكنك تحديد مدى معين له عن طريق الخاصية scope ، فمثلا لتخزين المتغير custAddr في المدى session يتم ذلك بكتابة السطر التالي

```
<c:set var="custAddr" scope="session"
value="${customer.address}"/>
```

ويمكنك الوسم <c:set> من تحديد قيم للخصائص بصوره منفردة ويتم ذلك تبعا للنموذج التالي:

```
<c:set target="bean-or-map-variable"
property="propertyName"
value="expression"/>
```

في النموذج السابق فإن الخاصية target يتم إعادة متغير من نوع map أو من نوع Java Bean فمثلا الكود التالي يقوم بتخزين الكائن Hash Map في كائن الطلب ثم يقوم بوضع قيمة في الكائن Hash Map عن طريق الوسم <c:set> (لمزيد من المعلومات عن الكائن Hash Map قم بمراجعه أي دليل للغة الجافا)

```
<%
  request.setAttribute("mymap", new
java.util.HashMap());
%>
<c:set target="${mymap}" property="myvalue"
value="123456"/>
<c:out value="${mymap.myvalue}"/>
```

ويمكنك هنا أيضا استخدام الصيغة الخاصة بوضع قيمة مباشرة للخاصية أو صيغه وضع قيمه للخاصية داخل جسم الوسم نفسه . وإذا تم تقييم التعبير إلى القيمة null وقمت باستخدام الخاصية var فإن المتغير يتم حذفه من المدى .

ويقوم الوسم <c:set> بإجراء أي تحويل مطلوب إذا كان نوع الخاصية عدد صحيح int أو عدد ذو دقه مضاعفه double فمثلا :محاولة الوسم

<c:set> تحويل قيمه نصيه string إلى int أو double وفيما يلي مثال للوسم <c:set>

```
<%@ page import="java.util.HashMap" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core"
%>
<html>
<body>
<%
  HashMap person = new HashMap();
  person.put("address", new HashMap());
  request.setAttribute("fred", person);
  ((HashMap) person.get("address")).
     put("street", "123 Blah Blahlevard");
%>
<c:out value="${fred.address.street}"/>
<c:set target="${fred.address}" property="city"
value="Blahse"/>
<c:out value="${fred.address.city}"/>
<c:set var="cityName">The city value is
   <c:out value="${fred.address.city}"/>.</c:set>
<p>
CityName=<c:out value="${cityName}"/>
<c:set target="${fred}" property="name" value="Fred
Fredrickson"/>
<p><c:out value="${fred.name}"/>
</body>
</html>
```

★ الوسم <c:remove>:

يقوم هذا الوسم بحذف متغير من مدى محدد أو من أول مدى تم إيجاد المتغير به فمثلا لتقوم بحذف المتغير custAddr من المدى session قم بكتابة السطر التالي

```
<c:remove var="custAddr" scope="session"/>
```

★ الوسم <c:catch>:

يتيح لك هذا الوسم أن تضع مجموعة من الوسوم داخل حلقه try-catch وأكثر استفادة من هذا الوسم التي تجرى مع التعبيرات التي تنتج القيمة NullPointerException فمثلا:

إذا كان لديك التعبير customer.address.street وكانت قيمة الخاصية address تساوى null فإنك سوف تحصل على الخطأ أو القيمة NullPointerException ، لاحظ فائدة الوسم <c:catch> في السطور التالية:

```
<c:catch>
  <c:out value="${customer.address.street}"/><p>
  <c:out value="${customer.address.city}"/>
  <c:out value="${customer.address.state}"/>,
  <c:out value="${customer.address.zip}"/>
</c:catch>
```

أما إذا أردت الوصول إلى كائن الخطأ نفسه exception فتستطيع استخدام متغير يحمل كائن الخطأ ولكن بعكس الوسم <c:set> فإنك لا تستطيع تحديد مدى له لأن المدى دائما يكون الصفحة نفسها Page .

مثال:

```
<c:catch var="exc">
  <c:out value="${customer.name.first}"/>
</c:catch>
```

لاحظ أن المتغير exc سوف يحمل كائن الخطأ إذا حدث.

✶ وسوم شرطيه

تسهل لك هذه الوسوم عرض نصوص في حالات معينه فقط .

✶ الوسم <c:if> :

يقوم هذا الوسم بتقييم التعبير ويقوم بإظهار القيم المحصورة داخله فقط إذا كانت نتيجة التعبير true فمثلا السطور التالية تقوم بإظهار قيمه متغير العنوان customer.address فقط إذا لم يكن يساوى null

```
<c:if test="${customer.address != null}">
  <c:out value="${customer.address.street}"/> <p>
</c:if>
```

ويمكنك أيضا استخدام المعامل empty لاختبار القيمة null

```
<c:if test="${!empty customer.address}">
  <c:out value="${customer.address.street}"/><p>
</c:if>
```

ويمكنك تخزين نتيجة التقييم في متغير عن طريق الخاصية var ويمكنك أيضا اختياريا تحديد خاصية المدى scope.

```
<c:if test="${!empty customer.address}"
var="custNotNull"
   scope="session">
  <c:out value="${customer.address.street}"/><p>
</c:if>
```

وإذا لم تقم بتحديد مدى فإن المدى الافتراضي يكون الصفحة Page ويمكنك تقييم تعبير معين ووضع القيمة بالمتغير بدون جسم خاص للوسم <c:if> كما يلي:

```
<c:if test="${!empty customer.address}"
var="custNotNull"/>
```

وهذا مساوي للتعبير التالي:

```
<c:set var="custNotNull" value="${!empty customer.address}"/>
```

★ الوسم <c:choose>:

يشابه هذا الوسم العبارة switch في لغة الجافا حيث يتيح لك الاختيار من ضمن عدد من الاختيارات المحددة وبينما العبارة switch لديها العبارة الملازمة لها case لأن الوسم <c:choose> لديه الوسم <c:when> والعبارة switch لديها أيضا العبارة default لتحديد الاختيار الافتراضي في حالة عدم توافق أي من الحالات الأخرى فإن الوسم <c:choose> له العبارة <c:otherwise> لأداء نفس الوظيفة ويتضح ذلك من المثال التالي:

```
<c:choose>
  <c:when test="${emp.salary <= 0}">
    Sorry, no pay for you!
  </c:when>
  <c:when test="${emp.salary <= 500}>
    Please come in on Friday...
  </c:when>
  <c:otherwise>
    Ok You can receive your pay check!
  </c:otherwise>
</c:when>
```

كما ترى من المثال السابق لعدم وجود عبارة مكافئه للعبارة else فأن الوسم <c:choose> يقوم باختبار قيمه أول وسم <c:when> وبالرغم أنه من الممكن أن تكون نتيجة تقييم عدة وسوم <c:when> تساوى true إلا أن أول تقييم true يتم تحقيقه فقط ، انظر المثال التالي:

```
<c:choose>
  <c:when test="${emp.firstName == 'John' &&
    emp.lastName == "Smith"}>
      Hello John!
  </c:when>
  <c:when test="${emp.lastName == "Smith"}">
    Hello, Mr. Smith!
  </c:when>
</c:choose>
```

في هذه الحالة إذا كان لديك الاسم John Smith فإنه يحقق كلا الشرطين ولكن الكود يقوم فقط بعرض "Hello John!" وليس "Hello, Mr.Smith"

✱ وسوم التكرار:

تستخدم هذه الوسوم لتكرار عدة سطور لعدد محدد من المرات أو عناصر كائنات معينه ويمكن لوسوم التكرار أيضا إرجاع متغير يحتوى على حاله التكرار والعنصر الحالي في مجموعة كائنات collections .

✱ الوسم <c:forEach>:

ويشبه هذا الوسم العبارة for الخاصة بلغة الجافا فيستطيع التكرار لعدد محدد من المرات أو خلال مجموعه من الكائنات وفى الحالتين تكون الخاصية الاختيارية var تحتوى على متغير الحلقة فإذا كان التكرار لعدد

محدد من المرات فإن متغير الحلقة يحتوى على رقم الحلقات التكرارية index أما إذا كان التكرار لمجموع كائنات collection فإن متغير الحلقة يحتوى على العنصر الحالي للمجموعة .

ولتقوم بتكرار عدد ثابت من المرات يجب أن تقوم بتحديد البداية والنهاية عن طريق الخاصيتين start وend على الترتيب ويمكنك أيضا تحديد الخاصية الاختيارية step التي تحدد مقدار الزيادة في رقم الحلقة لكل مره ، وافتراضيا يتم الزيادة بمقدار 1 فمثلا إذا كنا نريد تكرار كود معين 10 مرات فيتم تنفيذ ذلك كما يلي:

```
<c:forEach var="i" start="1" end="10">
  Item <c:out value="${i}/><p>
</c:forEach>
```

أما بالنسبة للتكرار لمجموعة كائنات فيجب استخدام الخاصية items لتحديد collection وتكون الخاصية items عبارة عن كود مكتوب بلغة التعبير Expression Language والذي يتم تقييمها إلى كائن مرجعي للواجهة Collection .

```
<c:forEach var="emp" items="employees">
  Employee: <c:out value="${emp.name}"/>
</c:forEach/>
```

ويمكنك أيضا في هذه الحالة استخدام عبارات التكرار الثابت start وend step إذا كنت تريد التكرار لعدد معين من العناصر داخل المجموعة .collection

★ الوسم <c:forTokens> :

يتعامل هذا الوسم مع النصوص المفصولة Tokens بحيث يقوم بإرجاع جزء من النص الأصلي حسب علامات الفصل المحددة ، فتحدد الخاصية items النص المراد فصله والخاصية delimiters تحدد قائمة بعلامات أو حروف الفصل التي سوف تستخدم ، وهذا الوسم يشابه الوسيلة java.util.StringToToknizer الموجودة بلغة الجافا .

مثال:

```
<c:forTokens items="moe,larry,curly" delimiters=","
  var="stooge">
  <c:out value="${stooge}/><p>
</c:forTokens>
```

ومثل الوسم <c:forEach> يمكنك تحديد البداية والنهاية ومقدار الزيادة للتكرار إذا كنت تريد التكرار لعدد معين فقط من المرات داخل أجزاء النص.

★ متغير الحلقة:

كلا من الوسمين <c:forEach> و<c:forTokens> لديهم متغير يسمى متغير الحلقة أوLoop status variable وفى كلا الوسمين تقوم بكتابة اسم المتغير مع الخاصية varStatus والقيمة الموجودة في هذه الخاصية تعرف الواجهة LoopTagStatus Interface وهذه الواجهة معرف بها الوسائل التالية:

LoopTagStatus Interface Methods	
Methodالوسيلة	Descriptionالشرح
Object getCurrent()	تقوم بإرجاع كائن التكرار الحالي (رقم التكرار أو العنصر الحالي في المجموعة collection
int getIndex()	يقوم بإرجاع رقم التكرار index بدء من الصفر بغض النظر عن البداية المحددة في الخاصية start فإن أول حلقه يكون رقمها 0
int getCount()	يقوم بإرجاع عدد مرات التكرار بدء من 1 فيكون دائما العدد يساوى الفهرس + 1
boolean isFirst()	يقوم بإرجاع true إذا كان هذا أول عنصر في المجموعة collection أو قائمة النصوص tokens أو أول رقم في حلقه تكراريه ثابتة
boolean isLast()	يقوم بإرجاع القيمة true إذا كان العنصر هو الأخير في المجموعة collection أو قائمه النصوص tokens أواخر رقم في حلقه تكراريه ثابتة
Integer getBegin()	يقوم بإرجاع القيمة المحددة في الخاصية begin
Integer getEnd()	يقوم بإرجاع القيمة المحددة في الخاصية end
Integer getStep()	يقوم بإرجاع القيمة المحددة في الخاصية step

★ وسوم خاصة بالروابط URL:

يوجد العديد من الوسوم التي تتعامل مع الروابط URL ويمكنها الوصول إلى موارد الويب المختلفة وتتيح لك الوسوم الكثير من الإمكانيات التي قد لا يدعمها المستعرض مثل cookies وتغيير الحروف encoding>

★ الوسم <c:url>:

يقوم هذا الوسم بتشكيل العنوان URL إلى نص حرفي وتخزينه في متغير وقد يقوم بتغيير الـ URL إذا دعت الحاجة . وتحدد الخاصية var المتغير الذي سيحمل الـ URL ويمكنك أيضا استخدام الخاصية الاختيارية scopr لتحديد مدى المتغير وإذا لم تحدده تكون الصفحة هي المدى الافتراضي أما الخاصية value فتحدد الـ URL المراد تشكيله.

```
<c:url var="trackURL" value="/page.html"/>
```

إذا كان المستعرض لا يدعم cookies فإن الـ URL الناتج يحتوى على معاملات session ID وقد يكون مسار URL مطلق مثل "http://www.sun.com/page1.xml" أو نسبى مثل "/welcome.jsp"

ويمكنك وضع معاملات في الـ URL عن طريق الوسم <c:param> داخل الوسم <c:url> والذي يحتوى على الخاصية name التي تحتوى على اسم المعامل والخاصية value التي تحتوى على قيمة المعامل .

```
<c:url value="/track.jsp" var -"trackingURL">
  <c:param name="myId" value="1234"/>
  <c:param name="typeOfView" value="summary"/>
</c:url>
```

★ **الوسم <c:import>:**

يشبه الوسم <c:import> الوسم <jsp:import> ولكن مع إمكانيات أكثر فعادة يقوم الوسم <jsp:import> باستيراد موارد من السيرفلت الحالي فقط ويقوم بإدخال المحتويات التي حصل عليها داخل صفحه JSP مباشرة ولكن الوسم <c:import> يستطيع وضع البيانات الخارجية داخل الملف أو إرجاع هذه البيانات كنص أو ككائن من نوع reader.

والخاصية الوحيدة التي يجب إدراجها هنا هي url التي تحدد المسار url للموارد المراد استيرادها وقد يحدد url على أساس مطلق أو نسبى .

ولكي تحصل على البيانات الموجودة بالمسار URL كنص يمكنك استخدام الخاصية var لتحديد متغير واختيار تحديد المدى عن طريق الخاصية scope والتي قد تكون page , request, session, application والمدى الافتراضي دائما يكون page . ويمكنك تحديد نوعيه الحروف التي سيتم عرضها عن طريق الخاصية charEncoding .

مثال:

الوسوم التالية تقوم بالحصول على بيانات URL كنص ثم تقوم بعرضه:

```
<c:import var="data" url="/mydata.xml"/>
<c:out value="${data}/>
```

ويمكننا تنفيذ نفس الوظيفة السابقة مع اختلاف واحد هو عدم استخدام متغير

```
<c:import url="/mydata.xml"/>
```

والخاصية varReader تحدد اسم المتغير الذي سوف يحتوى على بيانات URL ويمكنك هنا أيضا تحديد المدى عن طريق الخاصية scope ونوع الحروف عن طريق الخاصية charEncoding .

```
<c:import url="/mydata.xml" varReader="dataReader"
scope="session"/>
```

ويمكنك استخدام الوسم <c:param> ضمن الوسم <c:import> فيما عدا حالة استخدام الخاصية varReader للحصول على كائن reader وسبب هذا أن الوسم <c:import> يقوم بإنشاء كائن ready في الحال قبل أن يتم التعامل مع الوسم <c:param> .

وإذا احتجت أن تقوم بتمرير معاملات إلى الوسم <c:import> فقم باستخدام الوسم <c:url> لتقوم بإنشاء URL أولا كما يلي:

```
<c:url value="/myPage.jsp" var="tracking_my_page">
  <c:param name="trackingId" value="1234"/>
  <c:param name="reportType" value="summary"/>
</c:url>
<c:import url="${tracking_my_page}"/>
```

✶ **الوسم <c:redirect>:**

يقوم هذا الوسم بتغيير مسار المستعرض إلى URL آخر والميزة هنا عن استخدام الوسيلة response.sendRedirect هي أن الوسم <c:redirect> يقوم تلقائيا بتغيير المسار كما يدعم استخدام الوسم <c:param>

مثال:

```
<c:redirect url="http://www.sun.com"/>
```

مكتبة الدوال:

تحتوى مكتبة الوسوم JSTL على عدد من الدوال القياسية أغلبها يتعامل مع النصوص الحرفية وفيما يلي تفاصيل هذه الدوال:

★ الدالة fn:contains:

تعطى هذه الدالة القيمة true إذا كان جزء من نص موجود بنص آخر

وصيغة هذه الدالة هي:

```
fn:contains(original_string, substring)
```

حيث المعامل original_string هو النص المراد البحث فيه والمعامل substring هو النص الذي سيتم البحث عنه

مثال:

```
fn:contains("Alexandria", "exan")
```

تقوم الدالة في هذه الحالة بإرجاع القيمة true أما في الحالة التالية

```
fn:contains("Egypt", "Cairo")
```

فتقوم بإرجاع القيمة false

★ الدالة fn:containsIgnoreCase:

مثل الدالة fn:contains تبحث هذه الدالة عن جزء من نص داخل نص آخر وترجع القيمة true إذا تم إيجاده ولكن هذه الدالة تتميز بأنها تتجاهل حالة الحروف (صغيرة أو كبيرة) عند البحث وصيغة هذه الدالة هي:

```
fn:containsIgnoreCase(original_string, substring)
```

مثال:

```
fn:containsIgnoreCase("Alexandria", "EXAN")
```

ترجع الدالة القيمة true في الحالة السابقة ، وإذا كان المعاملان يساويان null يتم التعامل معه على أنه نص فارغ empty.

★ الدالة fn:endsWidth:

تعطى هذه الدالة القيمة true إذا كان نص معين ينتهي بحروف محددة وصيغه هذه الدالة هي:

```
fn:endsWith(original_string, end_substring)
```

مثال:

```
fn:endsWith("Alexandria", "dria")
```

تعطى الدالة في هذه الحالة القيمة true

★ الدالة fn:escapeXml:

تقوم هذه بتحويل النص إلى المماثل له بلغه XML صحيحة حيث تشبه تماما عمل الخاصية escapeXml الموجودة بالوسم <c:out> وتقوم باستبدال الحروف التالية:

<, >, &, ', "

بالحروف المقابلة للغة XML وهى على الترتيب:

<, >, &, ', "

وصيغة هذه الدالة هي:

```
fn:escapeXml(string)
```

مثال:

```
fn:escapeXml("<Hello>")
```

تعطى الدالة النص "<Hello>"

★ الدالة **fn:indexOf:**

تقوم هذه الدالة بإرجاع رقم أول حرف تم إيجاد حروف معينه لكلمة أو نص محدد وتشبه الدالة java.lang.String.indexOf الموجودة بلغة الجافا وصيغة هذه الدالة هي:

fn:indexOf(original_string, substring)

مثال:

fn:indexOf("Alexandria", "exan")

تعطى الدالة القيمة 2 لأن الدالة تبدأ العد من الصفر وإذا لم تجد الدالة النص المراد البحث عنه فإنها ترجع القيمة 1−

★ الدالة **fn:join:**

تقوم هذه الدالة بربط مجموعة نصوص في مصفوفة إلى نص واحد مترابط مع تحديد نوع الفاصل بين النصوص والصيغة العامة لهذه الدالة هي:

fn:join(strings[], separator)

مثال:

إذا كانت المصفوفة myWeb تحتوي على النصوص التالية "yahoo", "google", "sun", "Borland" فإن كتابة السطر التالي:

fn:join(myWeb, "/")

يقوم بإرجاع النص التالي :

"yahoo"/"google"/"sun"/"Borland"

وإذا كان المعامل الخاص بالحرف الفاصل فارغ "" فإن الدالة تقوم بإرجاع جميع النصوص بدون أي فاصل:

"yahoogooglesunBorland"

★ **الدالة fn:length :**

تقوم هذه الدالة بإرجاع طول النص المحدد وصيغة هذه الدالة هي

fn:length(collection_or_string)

وتنفيذ هذه الدالة على نص فارغ يؤدى إلى النتيجة 0

★ **الدالة fn:replace :**

تقوم هذه الدالة باستبدال جميع الحروف التي تم إيجاده ضمن كلمة أو نص معين بحروف أخرى محددة وصيغة هذه الدالة هي:

fn:replace(original_string, string_to_find, replacement_string)

مثال:

fn:replace("Alcendria", "ce", "xa")

وتكون النتيجة هي كلمة "Alexandria" كنتيجة للدالة.

★ **الدالة fn:split :**

تقوم هذه الدالة بفصل نص معين إلى مصفوفة نصية بناء على فاصل أو أكثر وصيغة هذه الدالة هي:

fn:split(string, token_string)

مثال:

fn:split("yahoo,google,lycos", ",")

تكون نتيجة العبارة السابقة مصفوفة تحتوى على ثلاث عناصر وهى "yahoo", "google", "lycos"

★ **الدالة fn:startsWith :**

تعطى هذه الدالة القيمة true إذا وجدت حروف معينة يبدأ بها النص المراد البحث فيه وصيغة هذه الدالة هي:

fn:startsWith(original_string, substring)

مثال:

fn:startsWith("Alexandria", "Alex")

تعطى الدالة في الحالة السابقة القيمة true.

★ الدالة **fn:substring :**

تعطى هذه الدالة جزء من نص عن طريق معامل للبداية ومعامل للنهاية والنص المقطوع لا يشمل آخر حرف محدد وصيغة هذه الدالة هي:

fn:substring(string, starting_pos, ending_pos)

مثال:

fn:substring("Alexandria", 2, 6)

تعطى الدالة في الحالة السابقة النص "exan" إذا كان معامل النهاية أقل من 0 أو أكبر من طول النص فيتم القطع حتى نهاية النص كله ، وإذا كان معامل البداية أقل من الصفر فيتم التعامل معه على أنه 0.

★ الدالة **fn:substringAfter :**

تبحث هذه الدالة عن حروف محددة داخل نص وتعطى الدالة النص التالي مباشره للحروف التي وجدتها وصيغة هذه الدالة هي:

fn:substringAfter(original_string, substring)

مثال:

fn:substringAfter("http://www.Borland.com", "http://")

تعطى الدالة النص "www.Borland.com"

★ الدالة **fn:substringBefore :**

تبحث هذه الدالة عن حروف محددة داخل نص وتعطى الدالة النص السابق مباشرة للحروف التي وجدتها وصيغة هذه الدالة هي:

fn:substringAfter(original_string, substring)

مثال:

fn:substringAfter("http://www.Borland.com", ":/")

تعطى الدالة النص "http"

★ الدالة **fn:toLowerCase** :

تعطى هذه الدالة النص المقابل للنص المحدد ولكن بحروف صغيرة وصيغة هذه الدالة هي:

fn:toLowerCase(string)

مثال:

fn:toLowerCase("AlExanDRia")

ويكون النص الناتج هو"alexandria"

★ الدالة **fn:toUpperCase** :

تعطى هذه الدالة النص المقابل للنص المحدد ولكن بحروف كبيرة وصيغة هذه الدالة هي:

fn:toUpperCase(string)

مثال:

fn:toUpperCase("AlexAndRia")

ويكون النص الناتج هو:"ALEXANDRIA"

★ الدالة **fn:trim** :

تقوم هذه الدالة بإزالة أي مسافات خالية من بداية أو نهاية النص وصيغة هذه الدالة هي:

fn:trim(string)

مثال:

fn:trim(" Hello ")

ويكون النص الناتج هو"Hello"

الفصل العاشر

قواعد البيانات

سنتعلم في هذا الفصل كيف نقوم باستخدام لغة الاستفسارات SQL والاتصال بقاعدة البيانات عن طريق البرنامج القياسي JDBC وأيضا استخدام وسوم مكتبة JSTL مع قواعد البيانات.

تمهيد:

المفهوم البسيط لقاعدة البيانات أنها مجموعة من المعلومات مخزنة في ملف ذو بناء خاص حتى يمكن الوصول إليها في أي وقت ، لذلك يمكنك أن تعتبر أن أي تخزين في ملف خارجي هو عبارة عن قاعدة بيانات.

ولكن قواعد البيانات الآن لها تركيب أكثر تعقيد وأكثر قوة وكفاءة لتخزين البيانات واسترجاعها ، فأي قاعدة بيانات يجب أن تدعم الخصائص التالية:

* تغيير البناء الداخلي والتوسع عند الحاجة.
* وسائل إدخال البيانات إليها .
* البحث والوصول السريع إلى البيانات المخزنة.
* وسائل التعامل مع البيانات المختلفة مثل التغيير والحذف وغيرها.

لغة SQL :

وتسمى لغة الاستفسار (Structure Query Language) وهى نتاج اللغة القديمة SEQUEL التي أنتجتها شركة IBM ، فهي لغة قياسية تتبع نظام ANSI (American National Standards Institute) وتتيح هذه اللغة جميع الإمكانيات اللازمة للتعامل مع قاعدة البيانات ، وتعتبر اللغة القياسية المستخدمة في العديد من قواعد البيانات الشهيرة مثل :

MS Access, DB2, Informix, MS SQL Server, Oracle, InterBase and Sybase

✶ أنواع البيانات التي تتعامل معها SQL:

تدعم لغة SQL جميع أنواع البيانات المعروفة التي يمكن تخزينها في قاعدة البيانات ، ومن أمثلة هذه البيانات:

binary, bit, char, datetime, float, integer, varchar

ويتم إجراء تحويل لأنواع البيانات من قاعدة البيانات إلى الأنواع المعروفة في لغة الجافا مثل String, Long, Float حتى تتناسب مع كائنات الجافا .

لغة SQL DDL:

وتسمى Data Definition Language وهي عبارة عن عبارات تستخدم لإنشاء وتعديل قاعدة البيانات ، وتشمل لغة DDL عبارات مثل CREATE و ALTER و DROP .

فمثلا العبارة CREATE تستخدم لإنشاء قاعدة بيانات والهيكل الداخلي لها مثل الجداول Tables والأعمدة والحقول لكل جدول . أما العبارة ALTER فتستخدم لتعديل الجداول والحقول ، والعبارة DROP تستخدم لحذف جداول أو أعمده من قاعدة البيانات .

وفيما يلي أمثله سريعة على استخدام العبارات السابقة:

CREATE DATABASE databaseName

تقوم هذه العبارة بإنشاء قاعدة البيانات بالاسم databaseName

CREATE TABLE tableName (columnName dataType,...)

تقوم العبارة السابقة بإنشاء جدول جديد بالاسم tableName ويتم تحديد أعمدة هذا الجدول ونوع البيانات الذي سيخزن به.

ALTER TABLE tableName ADD columnName dataType

تستخدم العبارة السابقة لتعديل جدول اسمه tableName بإضافة عمود جديد إليها .

ALTER TABLE tableName DROP columnName

تستخدم العبارة السابقة لتعديل جدول اسمه tableName بحذف عمود منها

DROP DATABASE databaseName

تستخدم العبارة السابقة لحذف قاعدة البيانات كلها.

DROP TABLE tableName

تستخدم العبارة السابقة لحذف جدول يسمى tableName

لغة SQL DML:

تختص لغة Data Manipulation Language أو DML بإدارة البيانات المخزنة بقاعدة البيانات ، وهذا يتضمن استخدام العبارات التالية:

INSERT, UPDATE, DELETE

تستخدم العبارة INSERT لإدخال بيانات إلى الجداول ويتم إدخال سجل في كل مرة ، أما العبارة UPDATE فتستخدم لتعديل محتويات الحقول أو الأعمدة خلال سجل أو عدة سجلات ، والعبارة DELETE تستخدم لحذف سجلات من الجدول ، وفيما يلي أمثلة لهذه العبارات:

INSERT INTO tableName (column1, column2,...) VALUES (value1, value2,....)

هذا السطر يؤدى إلى إدخال سجل جديد إلى الجدول tableName

UPDATE tableName SET column1 = new WHERE column1 = old

هذا السطر يؤدى إلى تغيير بيانات العمود أو الحقل column1 إلى القيمة new في الجدول كله إذا كانت قيمة العمود تساوى old فقط.

DELETE FROM tableName WHERE column1 = value

يؤدى السطر إلى حذف جميع السجلات التي فيها العمود column1 يساوى القيمة value.

لغة الاستفسار SQL:

بعد تسجيل البيانات وإضافتها إلى قاعدة البيانات فإنك قطعا سوف تحتاج لاستخراجها كلها أو بنائها بشروط محددة ، وهنا تأتى العبارة SELECT لهذا الغرض بالتحديد وتأتى عادة معها العبارة Where التي تحدد شروط استخراج هذه البيانات وفيما يلي أمثله:

SELECT * FROM tableName

الحرف * يعنى استخراج بيانات جميع الأعمدة أو الحقول بالجدول tableName

SELECT column1, column2 FROM tableName WHERE column2 = value

هنا يتم استخراج العمود column1 والعمود column2 بشرط أن يكون العمود column2 يساوى القيمة value

SELECT column1, column2 FROM tableName WHERE column1 = value AND column2 = otherValue

يشبه هذا السطر السابق ولكن مع جود شرطين لازم تحقيقهم .

ويفضل لمزيد من المعلومات عن لغة SQL أن تراجع مرشد قاعدة البيانات التي تقوم باستخدامها أو الرجوع الرابط التالي :

http://www.w3schools.com/sql/default.asp

اختيار قاعدة البيانات:

يمكنك استخدام قاعدة البيانات المجانية Cloudescape من إنتاج شركة IBM وتحميلها من رابط الشركة التالي:

http://www.ibm.com

وميزة Cloudescape أنها مكتوبة بالكامل بلغة الجافا ولها العديد من الإمكانيات المتقدمة ولكن إذا أردت استخدام قاعدة بيانات أكثر سرعه فيمكنك ذلك عن طريق قاعدة البيانات MySQL وهى أيضا مجانية ويمكن تحميلها من الرابط التالي:

http://dev.mysql.com/downloads/

والعبارات التي سنقوم باستخدامها يمكن تطبيقها على أي قاعدة بيانات متقدمه مثل SQL Server أو Oracle .

قم باستخدام برنامج محرر SQL الخاص بقاعدة البيانات المستخدمة (CView مع Cloudescape و SQL Plus مع Oracle) وقم بإنشاء قاعدة بيانات ثم اكتب الأمر التالي لإنشاء جدول يسمى People كما يلي:

```
CREATE TABLE People ( name VARCHAR( 75 ), age
INT )
```

لاحظ هنا أننا استخدمنا النوع varchar وهو النوع القياسي للنصوص string مع تحديد حجم هذا الحقل 75 حرف والنوع int وهو عدد صحيح لتخزين العمر ، قم الآن بإدخال بعض البيانات إلى الجدول عن طريق كتابة الاسم والعمر بالصيغة التالية:

```
INSERT INTO People( name, age ) VALUES ( 'anyName', '28' );
```

قم بتكرار السطر السابق عدة مرات مع اختلاف البيانات

استخدام JDBC:

قبل أن نتكلم عن كيفية الاتصال بـــ Cloudescape عن طريق JDBC يجب وضع الملف cloudscape.jar في الدليل WEB-INF/lib حتى يستطيع tomcat من تحميل ملفات تشغيل قاعدة البيانات.

وكلمة JDBC هي اختصار للكلمة Java Database Connectivity وهى مقابل ODBC لبرامج ويندوز وهى عبارة عن دوال API تمكنك من الوصول إلى البيانات الموجودة بقاعدة البيانات وتقدم العديد من الشركات ملفات تشغيل تسمى driver للاتصال بقاعدة البيانات وتعطى كلها نفس النتيجة .

ودوال API التي تحتويها JDBC تتيح لك ليس فقط الاتصال بقاعدة البيانات ولكن أيضا تنفيذ عبارات SQL على قاعدة البيانات والوصول إلى البيانات المرغوبة كما تدعم JDBC خاصية Pooling Connections أي إجراء أكثر من اتصال.

ولكي تتصل بقاعدة البيانات يمكنك إنشاء وصلة جديدة عن طريق العبارة التالية بالنسبة إلى Cloudescape

```
Class.forName("com.ibm.db2j.jdbc.DB2jDriver");
Connection con = DriverManager.getConnection(
  "jdbc:db2j:PeopleDB", username, password);
```

في السطر الأول تقوم بتحميل ملف التشغيل driver وفى السطر الثاني تقوم بعمل الاتصال بقاعدة البيانات عن طريق وضع مسار الاتصال وهو يشمل اسم المشغل db2j واسم قاعدة البيانات PeopleDB واسم المستخدم وكلمة السر لقاعدة البيانات .

سنقوم الآن بإنشاء الكائن statement الذي سوف يحمل أوامر SQL التي نريد تنفيذها:

```
Statement stmt = con.createStatement() ;
```

ويفضل أن نقوم بتخزين أوامر SQL أولا في متغير نصي وسوف نقوم بالنداء على المتغير كما يلي:

```
String sqlString = "some sql statements";
stmt.executeUpdate(sqlString);
```

أو

```
ResultSet rs = stmt.executeQuery(sqlString);
```

سوف تقوم باستخدام executeUpdate لتعديل أو إضافة بيانات مع العبارة UPDATE أو العبارة INSERT وإذا خمنت فالسطر executeQuery فيستخدم مع العبارة SELECT.

والخطوة الأخيرة دائما بعد الانتهاء من التعامل مع قاعدة البيانات هي إغلاق الاتصال حتى لا يتم استهلاك ذاكرة الكمبيوتر بالعديد من الاتصالات المفتوحة مع قاعدة البيانات ويتم ذلك عن طريق السطر التالي:

```
con.close();
```

ويجب عليك الآن تعلم المزيد عن دوال JDBC API عن طريق الرابط التالي:

http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/index.html

استخدام JDBC من خلال صفحات JSP والسيرفلت:

فيما يلي مثال على كيفية الاتصال من خلال صفحة JSP بقاعدة البيانات PeopleDB

```
<html>
<body>
<%@ page language="java" import="java.sql.*, java.io.*"
%>
<%
  Connection con = null;
  try
    Class.forName("com.ibm.db2j.jdbc.DB2jDriver");{
    con =
DriverManager.getConnection("jdbc:db2j:PeopleDB");
```

```
    Statement statement = con.createStatement();
    ResultSet rs = statement.executeQuery("SELECT *
FROM People");
%>
    <table border="1"><tr><th>Name</th><th>Age</th>
<%
    while ( rs.next() ) {
      out.println("<tr>\n<td>" + rs.getString("name") +
"</td>");
      out.println("<td>" + rs.getByte("age") + "</td>"); +
"</td>\n</tr>");
    }
    rs.close();
  } catch (IOException ioe) {
    out.println(ioe.getMessage());
  } catch (SQLException sqle) {
    out.println(sqle.getMessage());
  } catch (Exception e) {
    out.println(e.getMessage());
  } finally {
    try {
      if ( con != null ) {
        con.close();
      }
    } catch (SQLException sqle) {
      out.println(sqle.getMessage());
    }
  }
%>
</tr>
</table>
</body>
```

</html>

يقوم هذا المثال بالنداء على دوال JDBC الأساسية لطباعة كل السجلات المخزنة في الجدول People وفي الجزء الأخير من هذا الفصل سنتعلم كيفية تنفيذ نفس المثال بصوره أبسط باستخدام وسوم JSTL .

ملاحظات:

حتى تستطيع تجربة هذا المثال فإنك تحتاج إلى تحديد مسار قاعدة البيانات لخادم Cloudescape وافتراضيا يتم البحث في الدليل الحالي وهو bin/ ويفضل دائما وضع كل قواعد البيانات في دليل محدد ، فمثلا إذا وضعت قاعدة البيانات المسار التالي:

c:\myDb

فإن نص التوصيل يكوّن:

db2j.system.home=c:\myDb

وسيتمكن خادم Cloudescape من الوصول إلى قاعدة البيانات ، وبالنسبة إلى Tomcat فقم بتنفيذ السطر التالي قبل تشغيل Tomcat:

SET JAVA_OPTS="db2j.system.home=c:\MyDb"

استخدام مكتبة JSTL SQL:

مكتبة الوسوم SQL من أقوى المكتبات الموجودة في وسوم JSTL لأنها تعطى إمكانية تعديل البيانات واستخراجها عن طريق عبارات وسوم بسيطة ويمكنك استخدام هذه الوسوم مع وسوم مكتبه core لتكرار استخراج البيانات وعرضها.

★ الوسم <sql:setDataSource> :

قبل إجراء أي عمليات على قاعدة البيانات يجب أولا الإعلان عن محتوى البيانات datasource الذي سيتم استخدامه ، ويحدد عن طريق الخاصية var والخاصية الاختيارية scope ، وإذا لم تقم بتحديد متغير فإن datasource تصبح هي datasource الافتراضية لكل عمليات الوسوم الخاصة بمكتبة SQL وإذا حددت مثلا scope على أنه session فتصبح الـ datasource متاحة فقط خلال الجلسة الحالية current session.

ويمكنك تحديد datasource عن طريق الواجهة JNDI أو (Java Numbering and Directory Interface) عن طريق تحديد الخاصية datasource أو مسار JDBC واسم المشغل driver واسم المستخدم وكلمة السر والخواص لهذه المعاملات هي على الترتيب:
.url, driver, user, password

بعد إنشاء data source يمكنك التعامل مع قاعدة البيانات باستخدام JNDI.

★ الوسم <sql:query> :

يقوم هذا الوسم باستخراج البيانات وتخزين النتيجة في متغير الخاصية var ويمكنك تحديد المدى اختياريا عن طريق الخاصية scope ، وإذا لم يكن هناك data source افتراضيه فيجب تحديدها عن طريق الخاصية dataSource ويكتب مسار data source بلغة التعبير EL ، وإذا كان هناك متغير يحمل data source فيجب كتابة اسم المتغير محاطا بالرموز ${} .

وتحدد الخاصية sql كود SQL الذي سيتم استخدامه كما يمكنك كتابة هذا الكود في داخل الوسم <sql:query> نفسه ويمكنك أيضا تحديد الحد الأقصى للسجلات التي سيتم استخراجها عن طريق الخاصية maxRows وتحديد أول سجل يتم استخراجه عن طريق الخاصية startRow. كما يمكن استخدام الوسم <sql:query> مع معاملات وسيتم ذلك عن طريق الوسم <sql:param> أو الوسم <sql:dateParam> ويتم ذلك في داخل جسم الوسم <sql:query> وبالنسبة للوسم <sql:param> يتم تحديد قيمة المعامل عن طريق الخاصية value أو في داخل الوسم نفسه ، وبالمثل يمكنك تحديد قيمة للمعامل الخاص بالوسم <sql:dateParam> عن طريق الخاصية value أو داخل الوسم نفسه كذلك يمكنك تحديد نوع بيانات المعامل إذا كانت تاريخ date أو زمن time أو الاثنين timestamp والأخيرة هي الافتراضية . ويكون ناتج الاستفسار هو الواجهة Result التي تحتوى على السجلات التي تم استخراجها ، ومن الخصائص الهامة الخاصية getRows التي تعطى مصفوفة عبارة عن الكائن SortedMap فيمثل كل map سجل أو صف وكل key عمود أو حقل ، وهناك خاصية أخرى هامه هي getRowsByIndex التي تعطى مصفوفة أو مصفوفات لعدة كائنات .

أيضا يمكنك استخراج مصفوفة بأسماء الأعمدة عن طريق الخاصية getColumnNames واستخراج عدد الصفوف عن طريق الخاصية getRowCount ، ويمكنك تحديد ما إذا كان ناتج الاستفسار محدد بحد أقصى للصفوف عن طريق الخاصية isLimitedByMaxRows.

مثال: لعرض محتويات الجدول People

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core"
%>
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql"
%>
<sql:setDataSource
driver="com.ibm.db2j.jdbc.DB2jDriver"
   url="jdbc:db2j:PeopleDB" var="ds" />
<sql:query sql="select name, age from People"
var="results"
   dataSource="${ds}"/>
<html>
<body>
<table border="4">
<tr><th>Name</th><th>Age</th></tr>
<c:forEach var="row" items="${results.rows}">
  <tr><td><c:out value="${row.name}"/></td>
      <td><c:out value="${row.age}"/></td></tr>
</c:forEach>
</table>
</body>
</html>
```

يمكنك الآن أن تقارن بين هذا المثال والسابق له فالنتيجة واحدة في الاثنين ولكن المثال الأخير يوضح قوة وسهولة مكتبة JSTL .

★ **الوسم <sql:update> :**

يقوم هذا الوسم بتنفيذ عبارات التعديل وهى INSERT, UPDATE, DELETE واختياريا يعطي عدد الصفوف التي تم التأثير فيها في المتغير المحدد بالخاصية var كما يمكنك دائما تحديد المدى للمتغير عن طريق الخاصية scope اختياريا وإذا لم يكن هناك data

source افتراضية يجب تحديدها في الخاصية dataSource وطود SQL نفسه يتم وضعه في الخاصية sql في داخل جسم الوسم نفسه ، ومثل الوسم <sql:query> يمكنك تحديد معاملات عن طريق الوسمين <sql:param> و<sql:dateParam> .

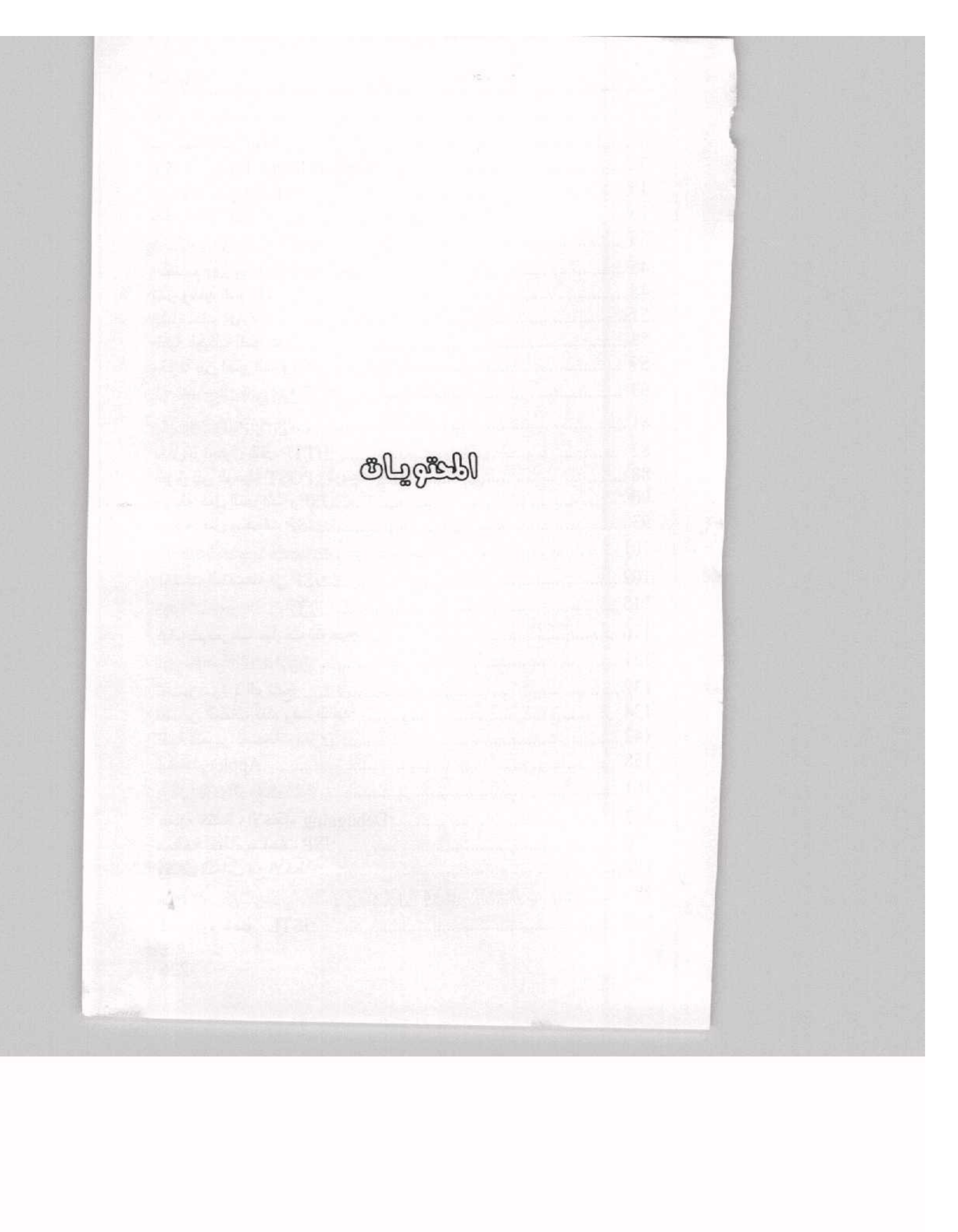
★ الوسم <sql:transaction> :

إذا لم تكن تعرف معنى مصطلح transaction في قواعد البيانات فهو يعني إمكانية تنفيذ مجموعة من العمليات المختلفة على قاعدة البيانات وحفظها في حالة نجاح تنفيذ كل هذه العمليات والرجوع عنها عند حدوث خطأ ما في أحد مراحل التنفيذ ويتم تحديد مستويات مختلفة isolation level (لمزيد من المعلومات قم بمراجعة كتب قاعدة البيانات التي تقوم باستخدامها) ويمكن تنفيذ transaction عن طريق وضع جميع الوسوم التي يجب تنفيذها داخل جسم الوسم <sql:transaction> ، وأيضا إذا لم يكن هناك data source افتراضي فيجب تحديدها عن طريق الخاصية dataSource وعدم تحديدها لأي من وسوم sql داخل جسم الوسم <sql:transaction> .

كذلك يمكنك تحديد مستوى transaction عن طريق الخاصية isolation والتي يجب أن تكون إحدى القيم التالية:

read_committed, read_uncommitted, repeatable_read, serializable

تم بحمد الله



المحتويات

طبعة سبتمبر 2005

رقم الإيداع
2005/14390
ISBN
977-17-2417-7



المركز الرئيسى : 11 شارع د/محمد رأفت – محطة الرمل – الإسكندرية
تليفون وفاكس : 4838326 (03)(+2)
موبايل : 0101634294 (+2) - 0123357844 (+2)
Email : info@egyptbooks.net
URL: www.egyptbooks.net